Department of Computer Science

Technical Report

Undergraduate Dissertation: An ontological approach to song scheduling for an automated radio station

Chris Day

Technical Report 2005-10 **October 2005**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

An ontological approach to song scheduling for an automated radio station

Chris Day

BSc (Hons) Computer Science

2005

An ontological approach to song scheduling for an automated radio station

Submitted by Chris Day

COPYRIGHT

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

……..

Abstract

In the competitive market of radio broadcasting it is imperative that radio stations tailor their content to suit their target audience and this process of targeting starts with the music policy. If a radio station gets the music wrong, then it risks alienating its listeners. This process of developing a music policy is never an easy process and what this project intends to do is implement a fully customisable song recommender system making use of semantic web technology to specify the relationship between songs and to allow the user to develop rules by which these songs are recommended. This customisation must not come at a price, however, and the system must continue to operate even if given bad or conflicting rules by the administrator. Any system that is broadcast-critical must be seen to be reliable whether it is for a national commercial station or a lowly student radio one.

Acknowledgements

Many thanks to my project supervisor Julian Padget for his support and guidance towards the direction of technologies new. Thanks too to the entire committee at URB (University Radio Bath) in particular David Mayo for imparting his intimate knowledge of URB's existing play-out system and finally a personal thanks to Gareth Gwynn, Lyndsay Fenner and Diet Coke™. Without them I would have probably left the library in a straight-jacket.

Contents

1. Introduction

In July 2004, the Student Broadcast Network announced that it was going into liquidation leaving student radio stations across the country without a music satellite service and no way to broadcast relevant content outside of presenter hours. Student radio, being an industry run by volunteers, do not have the resources of commercial radio stations to broadcast twenty-four hours a day and so with the demise of SBN another solution had to be sought. Many national commercial radio stations expressed an interest in filling the void, but student radio was not keen on the idea. The problem lies with station identity, for student radio is a niche market where its main selling point is that it can be different. In order to be different, it needs to have its own distinctive on-air identity, so many student radio stations have gone down the route of designing their own play-out computers to automate their on-air output whenever there are no volunteers available to do a show. URB (University Radio Bath) also opted for this, and its computer URB non-stop does an adequate job.

The problem is URB non-stop was never built with extendibility in mind, and as the pressures of the modern world demand integration, student radio is in danger of falling behind. This dissertation sets out to realise an autonomous song recommender system whereby the musical content of the radio station can be tailored using advanced categorisation of songs. The scope of this project is to create a flexible yet safe system that extracts the metadata from songs, and based upon this data makes decisions on the songs it plays. The system will be built with maintainability in mind, to separate fully the distinct parts, namely the decision-making part and the actual music player part. This prototype represents the base-level of the ultimate goal, to create a completely autonomous radio station.

With system integration in mind, the first part of the report will focus on the area of agent-orientated design and in particular its impact on the Semantic Web.

2. Literature Review

2.1. Introduction

The emphasis of this literature review will initially be split into two. First of all, the area of agent design will be explored, and then the Semantic Web will be explored. By Section 2.4, both of these concepts will be linked together in conjunction with additional work on Ontologies and existing projects related to this one.

2.2. Agent Orientated Programming

The term "Agent", in many people's eyes, is used rather loosely in Computer Science. This is largely down to the fact that concepts such as this are very hard to categorise absolutely, and in fact real world concepts like "Agents" only ever yield fuzzy categories (Franklin & Graesser 1996). As such, there can be no formal definitions as would be seen in a mathematical proof, only debatable descriptions. This is the closest I have seen to a clear description of autonomous agents:

*"An **autonomous agent** is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future."* – (Franklin & Graesser, 1996)

This statement was derived after analysis of several different definitions of agents. The important thing that distinguishes agents from ordinary computer programs is the idea of it pursuing its own agenda. That is to say, when a computer program calls a function, the commands within the function will execute precisely. When a computer program calls an agent to do something, the agent will decide whether it is in its interest to act on the request or not. This concept becomes clearer to understand if you take inspiration from biological theory, and make the analogy that the adaptive and autonomous quality of agents is comparable to living organisms (Steels, 1995). Indeed it could be said that humans are in fact just very complicated agents, and an object a thermostat is in fact just a simple one (Franklin & Graesser, 1996).

Agents in software have derived much inspiration from the Artificial Intelligence and Object Orientation communities. In terms of data encapsulation and message passing to execute methods, software agents can be said to have evolved from objects and so the mechanisms to achieve this are very similar. This is where the parallels end, however, since in order to be an agent, a role and an agenda is required. Object Orientation is simply a more structured approach to procedural programming and objects only ever respond to messages. To achieve autonomy and hence achieve basic agent-like behaviour, mechanisms must be added to analyse incoming messages and process them by first considering the internal state of the agent object (Guessoum & Briot, 1999).

The author has introduced the notion of agents acting in a role (much like a computer function) but only in accordance with its own agenda (unlike a computer function). There is one important feature of agents that needs to be emphasised. That is the agent's ability not only to carry actions that affects an environment, but also to sense the environment and to react to it accordingly. The term environment can apply to anything, physical or conceptual, that affects and is affected by the agent to some degree. Taking the previous example of a thermostat, the room in which the thermostat is located would be the environment. The thermostat senses the environment, and from that data decides whether to turn on the heating. Environments do not need to have a physical existence, indeed they do not need to only have one agent. In many environments there exist multiple agents performing different roles. To expand the example of a room, the thermostat may be one agent, the electric door may be controlled by another. This adds a new level of complication, not only must an agent sense an environment and react, but the agent may also need to know what other agents are doing. Let us assume the electric door agent keeps the door permanently open, in this case the thermostat may choose, given this data, to turn off the heating so as not to waste energy heating up an open room. This is a trivial example, but a non-trivial example could include a soccer match. The game is the environment, and the players and referee are the agents. This represents a complex example of a multi-agent system. There are agents on your team working with you towards a common goal; this is collaborative behaviour. There are also agents working together against you; this is adversarial behaviour (Stone, 1998). The other major factors that make this example non-trivial is as follows:

- You must communicate with other agents effectively since, at the high-level at least, you share a common goal with your team
- Accurate data on the environment is not complete as you cannot track the movements of all 21 players and the ball exactly at every instant. In real life soccer players can't see exactly what is going on so make decisions based on partial data.
- The game moves in real-time.

(*List adapted from* Stone, 1998)

There is a vast amount of debate regarding the scope and definition of agents. Nwana (Nwana, 1996) presents a slightly different set of views. Nwana attempts to condense down the wide ranging field of agents into 3 basic preconditions to agent-hood: They are the ability to learn, cooperate, and to be autonomous. To be an agent, it must have at least two of these properties. To be a smart agent, it must have all three. Although Nwana does not necessarily contradict Steels (1995), the approach taken to determine what is an agent is different. Steels concentrated on agents 'having their own agenda' whereas Nwana approaches the subject by studying an object's specific behaviour to determine agent-hood. The common ground in this debate is that agents do not execute commands blindly like a procedure or robot. An agent 'thinks' whenever a request is received and decides what to do, if anything. What is also agreed is that agents always must fulfil a role within a system or

environment. This is the reason why, when designing an agent-based system, a different technique should be followed.

In order to design a database, an entity relationship diagram is needed. This abstract model is then converted into something concrete (like tables in a database). The same can be applied to an agent-based system. One of the defining characteristics of agents and their relationship with each other and the system is the role the agent has. A role has associated with it two attributes: The permissions and rights and the responsibility (Wooldridge et al., 1999). The permissions and rights outline what the agent is able to do and conversely what it is not able to do; the agent's scope. The responsibility outlines what the agent should do in terms of functionality; the agent's domain. By creating a "role schemata", the design models are expanded to include the interaction protocols to show how the different roles interact. The combination of these is the mainstay of the analysis. From the analysis an agent design model consisting of the following can be created:

- The Agent Model: To document the various agent types used in the system when developing it. An agent type is best thought of as a set of agent roles.
- The Services Model: The equivalent model in Object Orientated programming would be a model representing methods. The subtle difference for agents is that agent 'services' are unavailable explicitly, but instead the services can be requested.
- The Acquaintance Model: Documents what messages can be sent between different agent types. This only covers potential communication links, not 'which messages will be sent and when'. This model is used to identify potential information bottlenecks and to evaluate how coupled a system is; how each agent is reliant on each other.

(*List adapted from* Wooldridge et al., 1999)

This provides an abstract design model of the agent system, but the limitations of this method is that the abstract model produced will not be sufficient to produce a system. The agent-orientated design model, as proposed, will rigorously detail what each agent should do and how it communicates. What the design fails to address is system-specific design models. In essence, the design describes what the agents should do, but not precisely how to do it. After producing the agent design abstract model, what then needs to be done is to use traditional software engineering techniques to convert these high-level concepts into lower-level abstract models. This techniques will depend on the programming languages used and the hardware associated.

My research into software agents is relevant to my project because the system I attempting to development could potentially be implemented in an agent-orientated fashion. The system is comprised of many parts, all requiring interaction. The interaction element is, of course, not the deciding factor, but the main engine of the project is the song recommender object. This object must act

autonomously and continue to recommend songs to be played by the playout object, and the object must respond to song requests. A simple robot would respond to requests to play them regardless, but this is not sufficient. The request may be inadvertently be breaking the rules, for example, if the song has already been played within 60 minutes. In this instance, the song request object would send a message to the song recommender asking if song x could be played. The song recommender will then decide whether the request is in keeping with the rules and will accept or reject the request. I would interpret this as the song recommender 'having its own agenda' as defined by Steels (1995); Guessoum & Briot (1996). Taking inspiration from what the agent community can offer to my final project design could be beneficial in designing a maintainable and robust system.

2.3. Introducing the semantic web

2.3.1. Definition

One aspect of agent orientated design that I failed to mention in any great depth is the mechanism that software agents use to communicate with each other. The interactions between robots and agents in multi-agent systems are key factors in the success of a system. In order for communication between agents to be productive, they must share a common language. This does not mean that the language for each agent must be identical, but as a subset of both agents' languages must be this 'common language' where the vocabulary and the semantics are the same. Developers of the World Wide Web have realised the massive problem that this represents. Here I introduce the Semantic Web.

The Semantic Web has been cited as the next generation of the World Wide Web by Berners-Lee and others (2001). Berners-Lee et al. (1992) first introduced the World Wide Web as the 'information universe'. W3, as it is also called, has meant the Web can stretch seamlessly from personal notes on a local workstation to mainframe databases on the other side of the world (Berners-Lee et al., 1992). With W3, using simple text searches, anyone connected via a telephone line could access any information from around the world. The flexibility and diversity of W3 is both the greatest asset and the greatest weakness. HyperText Markup Language was designed so that the same information could be formatted in many ways, but with agents and embedded systems ever evolving, so is the need for autonomous systems. The major problem is HTML and most of the Web's content today is that the data presented is designed for humans to read (Costello et al., 1999), not for computer programs to manipulate meaningfully (Berners-Lee et al. 2001). The Semantic Web aims to rectify this problem by adding meaning to web pages. Whereas somebody's name would be often seen on the top of a W3 page, in a Semantic Web page there would be a specific tag to say 'Authored By'. W3 attempted to improve its semantics by the introduction of HTML 2.0 (Berners-Lee & Connolly, 1995; cited by Luke et al., 1996) which included REL, REV and CLASS subtags, and the META tags. These mechanisms were rather weak and were really only used for document meta-information such as keywords. The mechanism was hindered by the fact that relationships could only ever be formed

by creating hyperlinks (Luke at el, 1996). Simply extending HTML to include semantics was not going to satisfy the vision of the Semantic Web. A new markup language had to be created.

2.3.2. Introducing XML and RDF

Extensible Markup Language, abbreviated to XML, provides the flexibility required to begin to define semantics of a web document. With XML, it is easy to create a custom tag set which means instead on relying on the rather inflexible HTML 2.0 META tags, tags can be created that can help identify the information useful for search and retrieval of the document (Usdin & Graham, 1998). Indeed the HTML language tags are fixed, meaning that domain-specific data like a patient ID for a hospital application could not categorised (Costello et al., 1999). It is important to note, however, that the emphasis in the sentence 'XML can help identify information for search and retrieval' (Usdin & Graham, 1998), should be on the 'help identify'. XML does not provide semantics to the document, but is a useful data definition language upon which semantic assertions can be made. What needs to be built upon this is a data description language that not only allows the web page to declare what data it contains, but also the relationships between them. Decker et al. (2000) back up this view and clarify that XML "address only document structure". Decker et al. (2000) goes on to suggest that Resource Description Framework (Klyne & Carroll, 2004), abbreviated to RDF, is a suitable language to layer on top of XML. RDF has a web-orientated emphasis so will often reference other web files, but RDF is not specific to any type of resource. When addressing resources, RDF uses Universal Resource Identificators (URI) which should not be confused with Universal Resource Locator (URL) used by web pages to find other pages. While RDF does use XML to exchange descriptions of Web resources, the resources being described can be of any type, including XML and non-XML resources (Brickley & Guha, 2000) like sound files. In the area of multimedia files, the primary domain for my project, the non-specific flexibility offered by the RDF/XML combination is a positive point. What RDF achieves is a data model by adopting a triple structure as illustrated in *Figure 1.0*.

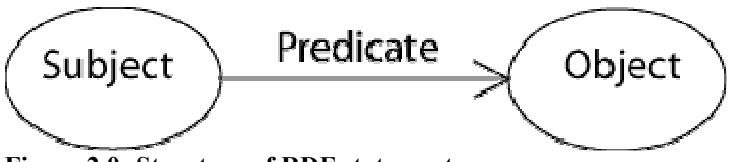


Figure 2.0: Structure of RDF statements
Adapted from Klyne & Carroll (2004)

An example of an RDF triple would be:

…
<#chris> <#age> 20
…

The subject in this case is <#chris>, the predicate is <#age> and the object is '20'. Adding this triple would mean that an agent processing the .rdf file would know

that ‘chris’ has an ‘age’ property of 20. <#chris> and <#age> would have to be in this case defined elsewhere in the document, but 20 is a literal. By having a series of triples, further information can be elicited.

…
<#chris> <#age> 20
<#chris> <#wrote> <http://www.bath.ac.uk/~cs2ccd/PROJECT>
…

Now not only is the fact that <#chris> is 20 years old, but also wrote the afore mentioned page. The more triples are adding to the document, the more semantics agents will be able to derive from it. The use of RDF/XML layering has recently received a W3C (The World Wide Web Consortium) Recommendation, however this does not mean that the method is agreed by all. Carroll & Stickler (2004) are highly critical of the current use of RDF/XML layering. Carroll & Stickler (2004) present an alternative XML syntactic structure to RDF (entitled ‘TRiX’). Many of syntactical problems have been fixed after a major clean-up of the syntax by the W3C and it is possible to have good interoperability there is still the issue that RDF is hard to validate when embedded in XML documents (Carroll & Stickler, 2004). On balance, Carroll & Stickler (2004) recognise that RDF is a concise language and some of the syntactical features make it ideal when constructing Ontologies. Haustein & Pleumann (2003) propose that the concept of Semantic Web itself is not without problems. At present, the Semantic Web is still in the concept stage, and there is not a sufficient base of RDF-annotated pages. This means the current benefits to smaller institutions or individuals, who can ill-afford the additional work, are minimal. These people are required for the Semantic Web project to reach its ‘critical mass’ (Haustein & Plaumann, 2003). Although not offering a solution to the problem highlighted by Haustein & Plaumann (2003), Grau (2004) has put forward a possible simplification to the current Semantic Web architecture to solve some of the layering problems highlighted by Carroll & Stickler (2004). The proposed simplification will mean better layering, especially when incorporating Ontology-based languages such as OWL (described later). Instead of the Ontology language sharing a common syntax with RDF and merely offering a semantic extension, the Ontology language will extend both the syntax and semantics of RDF. The result would be reduced syntactic expressiveness of RDF but would mean the architecture would be better designed (Grau, 2004). Further research on Ontological languages will be conducted later.

2.3.3. A multimedia-specific alternative to RDF: MPEG-7

Multimedia and related information on W3 is vast, but much akin to the problem with web pages; most of the information is not readily machine readable. As such, research has been conducted into how multimedia can fit into the Semantic Web. I have already explored RDF which is the general descriptor of resources, but there is a much more multimedia-orientated metadata language being developed; this is MPEG-7. The goals of MPEG-7 is vastly different to the audiovisual encoding standards MPEG-1,2 and 3, since MPEG-7 is not a new method for compression but a mechanism for extracting and exchanging features and metadata associated

with multimedia files (Crysandt & Wellhausen, 2003). MPEG-7 is not just a re-write of the already common ID3-tags found in many MPEG-1 layer 3 music files. Aside from the expected metadata such as artist, song title, genre, and the like, MPEG-7 supports data to do with the actual sound wave itself such as Audio Spectrum Spread (used to differentiate between tone-like and noise-like sounds), Audio Spectrum Centroid (determines whether the power spectrum is dominated by low or high frequencies) and the like (Crysandt & Wellhausen, 2003). Going into any detail about the nature of these calculations is beyond the scope of this literature review. This data can be used to automatically elicit information about the waveform without having to use advanced sound evaluation techniques since this data will be encoded automatically when the audio is recorded (*see figure 1.1*).

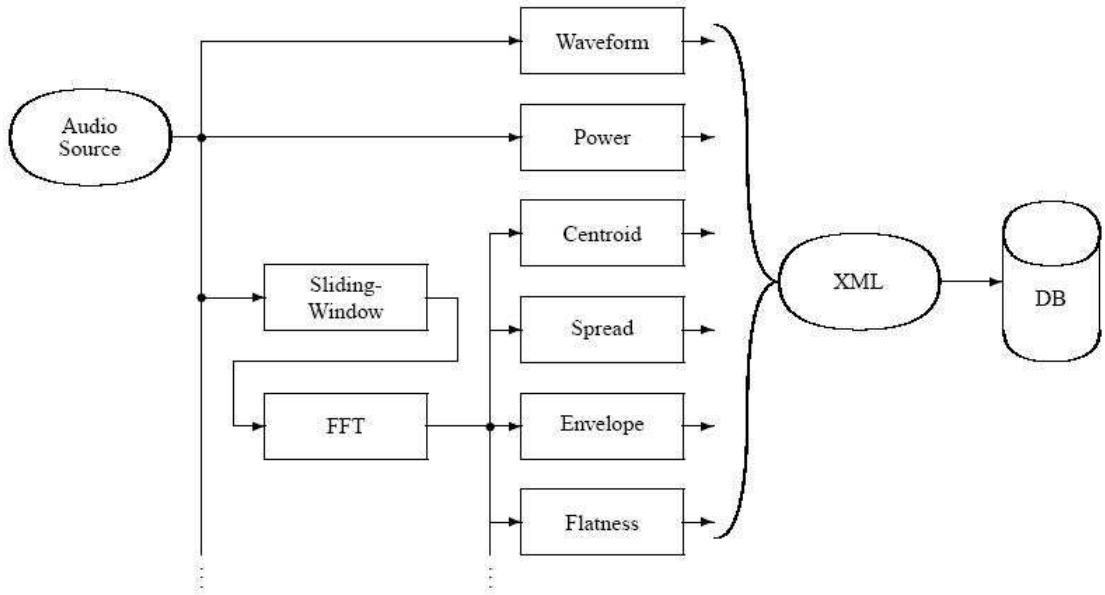


Figure 2.1: Automatic feature extraction of MPEG-7
(adapted from Crysandt & Wellhausen, 2003)

Further adaptations to the MPEG-7 framework will also include beat-analysis which would mean automatic DJ programs could seamlessly mix audio clips together with minimal effort. MPEG-7 is designed to be the complete content management solution and is likely to become a standard in the near future. As you can see from the diagram, the MPEG-7 standard extends from an XML syntax much like RDF described before. In order for MPEG-7 to be considered, however, more research will be needed to establish the practicalities of using MPEG-7.

2.3.4. Problems with the semantic web

In this subsection, there are two general method to represent multimedia metadata; RDF and MPEG-7. The common element in both is the basic XML syntax system. In this subsection about the Semantic Web, there has been no discussion on Database Management Systems and their role. As XML technology is being developed, so is the concept of XML databases. It is important to note that XML

databases are not being developed to replace Database Management Systems (DBMS). XML and database technology are more complementary than competitive (Costello et al., 1999). XML and databases are trying to achieve different things entirely. Databases, although vital to the Web as we know it, could not make up the basis of the Semantic Web. The ethos of the Semantic Web is for data to be simple and to be portable. XML provides a data structure that is semi-structured (Costello et al., 1999) whereas DBMS systems are organized around rigid relational tables. For this reason, databases and other rigid-structured data models could never be used in the World Wide Web, this would be the equivalent of enforcing the rule that no hyperlink is allowed to be 'broken' or link to a page that does not exist. Such a rule for something as diverse as the Web can not be enforceable.

2.4. Agents and the Semantic Web

In this literature review, there have been two topics of discussion, Agents and the Semantic Web. Agents were discussed in relation to the potential structure and method of the code, whereas the Semantic Web is very much in the domain of being able to distribute information. These topics are by no means mutually exclusive. In Section 1.2 of this review, the problem of agent vocabulary was discussed. In order for agents to collaborate with each other to monitor and modify the environment, they must share a common vocabulary. Chen et al. (2003), while developing the TAGA (Travel Agent Game in Agentcities) system strongly advocates the use of Semantic Web technologies in the agent developed. The main reason cited was that it this use 'improves interoperability between agents'. In other words, the ability to exchange and use information is easier. It makes sense, since what the Semantic Web is trying to achieve (namely a world-wide network of machine readable data) with technologies such as RDF, is precisely what the agent community requires. RDF is a good fact-stating language, but RDF alone is not sufficient to produce sufficient logical support for agents. RDF is good at describing resources, like music files, but what it lacks is the ability to declare classes and specific logical combinations of these classes such as union and intersection; this is where OWL is introduced (Horrocks et al., 2003).

2.5. Introducing Ontologies and OWL

OWL (Bechofer et al., 2004) stands for Ontology Web Language, to understand how OWL differs from RDF, the concept of Ontologies must first be explored. In philosophy, the word ontology means the study of things that exist. This generic and all-encompassing term has been borrowed by computer science to mean the structure of agreed knowledge or semantics within some domain or environment (Chandrasekhan et al., 1999; Spyns et al., 2002). This definition can be easily applied to the notion the Semantic Web. RDF and other 'fact-stating' languages are not sufficient for something as complicated the World Wide Web. The Web, due to the diversity of users and publishers, will have countless RDF classes which essentially mean the same thing semantically but would be processed as separate entities by a computer agent. The purpose of OWL, as an ontology

defining language, would be to bring together all of these classes that are syntactically dissimilar, but semantically identical. In the same way the Semantic Web can not rely on 'fact-stating' languages alone, neither can the field of agent software. For example, music recommender and play out software, RDF (or MPEG-7 either can be used interchangeably) would be sufficient to describe songs, their artist, titles, genres, categories and any other meta-data associated with them. In a song recommender software situation, however, knowing what genre each song is, is simply not sufficient. The software may have rules such as 'do not play the same song twice in the same hour', or 'in this hour do not play songs from category A'. Such statements can not be expressed as facts; they are rules. Rules, and their evaluation, require a Description Logic Language in order to assert facts. OWL-Lite and OWL-DL (two variants of OWL that are described soon) can be viewed as expressive Description Logics (Horrocks et al., 2003). Hendler (2001) advocates the union of the Semantic Web technology and agent-orientated design stating one of the reasons being the ease of communication between agents since they share a common language (in the shared ontology).

As stated on the W3C Recommendation (Bechofer et al., 2004), there are three distinct variants of OWL. They are called OWL-Lite, OWL-DL and OWL-Full. Lite is a subset of DL which in turn is a subset of Full. The choice of the variant to use really depends on the application of the ontology, as this choice like many others is a trade-off. OWL-Full offers the greatest compatibility with the RDF layer as it allows free mixing of OWL with RDF schema, and like RDF schema, does not enforce the strict separation of classes (Bechofer et al., 2004). On the hand, OWL-DL may put a constraint on the use of RDF (namely disjointness of classes, properties, individuals and data values) but the adding benefit is the OWL-DL can be reasoned using inference engines. Unlike OWL-Full, OWL-DL is decidable and the same applies to OWL-Lite (where further constraints are applied). For the purposes of agent software, OWL-Full would be inappropriate, since agents require Ontologies to be reasonable.

OWL is very much 'work-in-progress', and as such the support for automatically parsing OWL (vital for agents) is still under development. At present there are only two reasoning engines for OWL, RACER (Haarslev & Moeller, 2003) and JENA (Carroll et al., 2003). RACER is the engine used in 'Protégé', one of main applications that can be used to develop Ontologies. More research is required to provide better support for agent software.

2.6. Related completed work

There are many projects in the same domain as this one, but the one that is most relevant to build upon was implemented by Papadakis & Douligeris (2002). Papadakis & Douigeris design a system that automatically extracted the IDE tags from MPEG 1 layer 3 files and created a metadata database in XML. This project recognised the problem surrounding data about multimedia files, drawing Napster and Gnutella as prime examples as their systems "relied on the filenames of the mp3s". As a digital library, it works well, but the only criticism would be that the use solely of XML and not the other Semantic Web languages would limit what

could be done. Indeed, in terms of this project, use of XML would not be sufficient.

2.7. Summary

This literature review has attempted to research the domain of agent design and the Semantic Web with an emphasis on Ontologies, the linking factor between the two concepts.

In agent design, the most important aspect for agents is that they are not simply dumb robots (or objects in the case of object orientated programming). Agents receive messages to methods, but instead of executing the method verbatim, the agent will analyse the request according to its internal state (Guessoum & Briot, 1999). In short Agents have their own agenda and will act autonomously.

Another key factor regarding agents was the ability to communicate between agents; collaborative behaviour. The review found that having a common language between the agents and the environment was the only way meaningful interaction can take place. It was established that the Semantic Web can help in this process. The Semantic Web being the attempt to make the web, machine understandable; the technologies of which are also usable for agents. The conceptual overlap between agents and the Semantic Web is large. RDF (fact declaring language for generic resources) and MPEG-7 (fact declaring language specific to multimedia files) were also compared and contrasted.

The final area addressed by this review was the area of Ontologies. A new Ontological language (OWL) designed to layer over RDF as part of the semantic web is still in the research phase, but support is becoming more available to parse it. Ontologies are needed to add more semantic meaning to the fact-declaring languages by adding notions such as union and intersection between classes. More research into the practicalities of OWL is needed as well a more in-depth analysis of OWL as a language.

3. Requirements Elicitation

3.1. Methodology

In order to understand how to proceed in designing the system, it was important first to understand the problem. The application of this system was to a very specific environment, namely a radio station. Throughout this section University Radio Bath (URB) will be used the main case-study and all the requirements and analysis was based on discussions that took place with former station manager David Mayo, the expert user. Mayo is very used to dealing with music play-out systems and is one of the IT technicians in charge of URB's existing automated play-out software, URB Non-Stop, which was originally developed by Mark Chappell. URB Non-Stop was the second main focus of this process, as its faults and successes drove the analysis of the problem forward. The elicitation process is divided in sub-sections, each covering an area of the system.

3.2. Music Metadata

3.2.1. What is stored

Music metadata was the crux of this problem as song recommendations are based upon this information and so there needed to enough information about the songs in order to carry this out. From the expert user's report, it was indicated that the following will need to be stored by the system about each song:

- The artist
- The title
- The album
- The category
- The year of release
- The length

It is worth noting that 'The Category' attribute is not the same as genre, this important distinction is discussed later in this section.

3.2.2. Where it is stored

The question came down to a basic choice of storing the metadata in the same location as the data or storing it elsewhere. Generally it is much better not to separate the metadata and the data because of the risk of losing synchronisation. This became a prudent question when considering the problem of file moving or renaming as this means the file (essentially the 'data') is no longer addressable by the same reference. If the metadata is located is another place, this change in reference may not be updated, however if the metadata is stored actually within the file itself there is no danger of this happening. It was clear that the safest way to preserve the integrity of the data was to make sure that the metadata is stored at

the source. The integrity of this metadata is important to prevent songs being recommended to play when actually they no longer exist, which could have serious implications for the play-out engine which may not be expecting to have to deal with an unplayable file.

3.2.3. How the existing system addresses this

URB non-stop, the existing system, adopts the method of storing the files in directories meaning that conceptually the metadata is in indeed stored in the same location as the file. The artist and title of the song is taken from the filename in the form: `artist - title.mp3` and the file lengths are taken by extracting tags from the mp3 files themselves. URB non-stop does not use a database to store a table of songs so although this method is rather crude, it does enforce referential integrity of the data.

3.2.4. Requirements elicited

- The music meta data must contain:
  - The artist
  - The title
  - The album
  - The category
  - The year of release
  - The length
- The music metadata used for song decision making must be stored within the music data file to preserve referential integrity

3.3. System structure

3.3.1. Combining the components

One thing that the expert user's report made reference to is the requirement to ensure that there is autonomy between the different modules. This meant there must be a distinct separation between them and that they are not reliant on each other. The rationale behind this requirement was because if they can operate independently and only share enough data for all the modules to carry out their tasks, then the system could be designed so that the different modules are physically on different machines thus creating a certain level of redundancy. This means that if one module fails, the other components will be able to cope independently. An additional supporting argument for this requirement would be the ease at which modules could be substituted for better ones. For example, if the system is comprised of autonomous modules and the song recommender engine is working well but the radio station would like to upgrade the play-out engine, this process is made much easier because the only thing that the administrator would need to consider is whether the new play-out engine's interaction with the shared data structure mirrors that of the old play-out engine. The recommender engine would not have to be touched at all.

This distinct separation requirement brought with it a series of new requirements, for if the core data was being shared between distinct modules then the data both needed to have persistence (in other words whenever the data is accessed and changed by one module, this change is automatically reflected in the other modules' view of the data) and needed to allow concurrency (as many modules may access the data at the same time, the mechanism by which this is done needs to work). A failure to do either would result in the system being in a serious asynchronous state.

3.3.2. Requirements elicited

- Must have a distinct abstraction between the recommender engine, the request engine and the playout simulator
- The song data must be shared available to all modules
- The song data must be persistent
- The song data must allow concurrency

3.4. Song selector algorithm

3.4.1. Deterministic or random

This question was raised because David Mayo stated that it is undesirable to have a song scheduler that displays completely predictable behaviour. The justification behind this statement was because some listeners would listen at regular times during the week and if the scheduler were to be too predictable then the same songs would end up getting played at the same time period. The chances of this occurring are unlikely, since it would need exactly the right number of songs to produce this effect; however it was significant enough for consideration. Therefore it was sensible to stipulate that although parts of the recommender engine may operate deterministically, there should at least be some random element even if it is a weighted random.

3.4.2. Ensuring it is real-time

The efficiency of the song recommender engine only came into question really if the required computational time is longer than the songs actually being played. This was one requirement that was non-negotiable as failure to satisfy it would mean the play-out engine would have periods of time where there was nothing being played. For radio stations whose broadcasting licences depend on them providing a reliable service, this matter was of utmost importance.

3.4.3. Taking requests from internet

One of the best ways of tailoring the radio station music output to suit the audience is to allow the listener to request a song. This is a standard facility

offered by radio stations when there is a live presenter in the studio, but it would much enhance the credibility of the station if this service was also offered outside of live presenter hours. There should be a limiting factor to this as there is a potential that one listener may be to hi-jack the radio station and use it as their own personal jukebox. This would have the potential of alienating other listeners, therefore it was only sensible to impose some sort of upper limit beyond which no other requests will be accepted.

3.4.4. How the existing system addresses song selection

URB non-stop has an entirely random approach to selecting songs from a given category. Regardless of any attributes, as long as the song belongs to the category that is due to play next then each song has an equal chance of getting played. Non-stop also has no problem with its recommender engine taking longer than the length of songs as the non-stop works out the entire hour's play-list beforehand. The final facility, requests from the internet, is not possible using the URB non-stop system for the simple reason that the entire hour's play-list is predetermined so there is no dynamic way of adding songs that the listeners want to hear. This gave rise to another requirement, namely that songs should be recommended on a by-need basis and not to generate an hour-long playlist to allow the requests to be played as quickly as possible.

3.4.5. Requirements elicited

- The song selection process must not be entirely deterministic, there must be a random element to it
- The song recommender engine must operate in realtime and be able to recommend songs faster than it takes to play them
- The system shall be able to take listener requests as long as the number of requests has not exceeded the request limit
- The system shall recommend songs on a by-need basis only

3.5. Repetition of songs

3.5.1. Ensuring variety

One of the highest sources of complaints received at URB regards the repetition of songs. Part of this problem is due to the play-list system adopted and this is discussed later, but also an important factor is way songs played are recorded and how this information is factored in when making a song recommendation. David Mayo suggested that there should be a period of time after a song has been played where that song is banned and not considered for recommendation. Mayo had stated that this requirement was a high priority not only because it may annoy listeners but also may violate the copyright agreement (PPL, 2005) that URB has in place for its web broadcasting. Further to the rule that prevents songs being repeated, it was suggested that songs that have been least recently played should get priority when a song needs to be recommended. The reasoning behind this

notion is because the content is then much likely to be varied which makes for a better listening experience.

3.5.2. How the existing system addresses this

URB Non-stop has scheduling rules in place to ensure that songs are never played twice in a given hour period. These rules are also extended to cover not only individual songs being played twice, but also artists as well. This extended rule is just as important as the rule that covers individual songs as not considering this eventuality at all could lead to several songs by the same artist being played in a row. This assertion was not strictly true, as non-stop does not apply this rule over a time period; rather it applies it over the last fifteen songs played. This achieves the goal it was intended to do, however it is much more intuitive for the user to think in terms of time not number of songs since songs come in quite varying lengths. The longest songs can be six minutes whereas the shortest songs can be as short as one and a half minutes long. On the strength of this it was decided that the user-defined repetition threshold should be defined only in terms of time, not number of songs.

3.5.3. Requirements elicited

- Songs must not be repeated before the minimum period set by user has elapsed
- Songs by the same artist must not be repeated before the same minimum period has elapsed
- The minimum repetition value must be set based on time rather than number of songs played.
- When recommending songs, priority should be given to songs less recently played

3.6. Category inheritance

3.6.1. One size does not fit all

The traditional view that songs should be categorised into genres is unfortunately an over-simplification of the situation. Radio stations do not only want to decide the type of music they play based solely on genre, because other factors such as how recent or how popular/mainstream the music is are just as important. This all means that songs can and often will belong to more than one category. From the discussions with the expert user, it was agreed that the following categories will be required:

- Rock
- Dance
- HipHop
- ChillOut
- DnB

- SoulnRnB
- Pop
- Alternative
- Recent
- Classics

It might have been enough to say that songs should be allowed to belong to more than one category, but the major objection to this was that this structure-less approach could lead to logical contradictions such as a song being both a 'Recent' and a 'Classic', and it is such contradictions that are inherent in the design of URB Non-Stop, an existing system which serves the same purpose as this project.

3.6.2. How the existing system addresses this

URB non-stop's method to categorising songs is to use the physical directories the files are stored in. Each 'category' is explicitly defined in text-files by stating which directories belong to which category. Each directory could feed into more than one category, and each category could have more than one directory feeding into it, so it is a many-to-many relationship. This is a rather convoluted way of saying that each song can belong to more than one category. There is one major flaw with this method; however, as in order to be in more than one category, the song needs to be in more than one directory, thus meaning the song must be duplicated to achieve this.

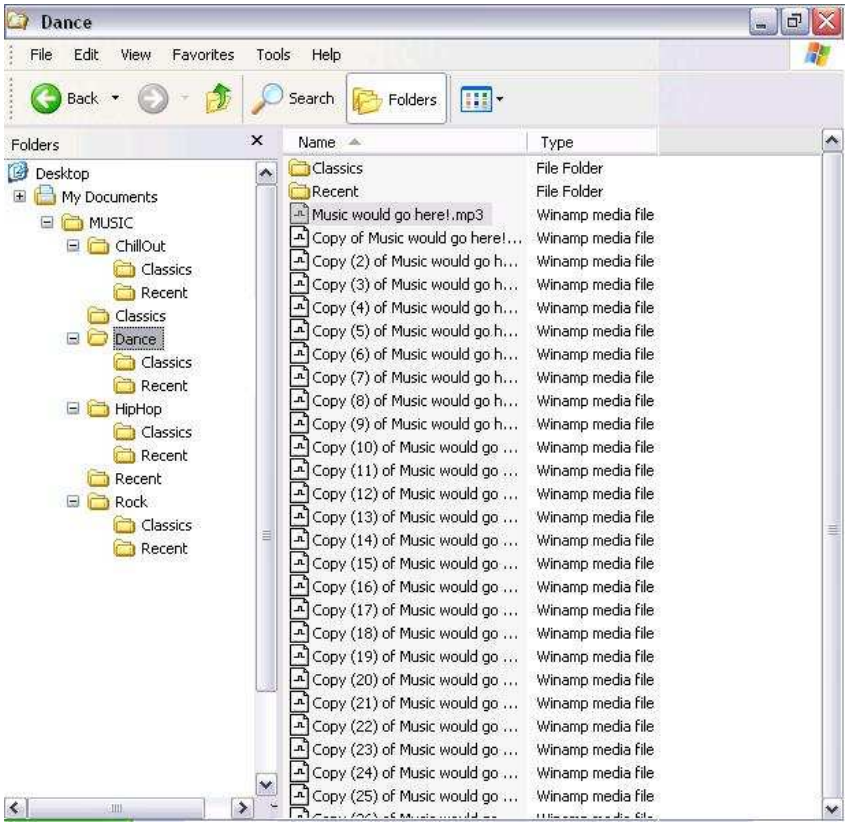


Figure 3.1: The music directory structure of URB non-stop

This naturally is rather wasteful of disk space and has the unfortunate side effect of being unable to determine the categories for any given song. It can only determine the list of songs, given a category not vice-versa. There were some merit in the method, however, as the way categories are mapped to directories provided some clue as to how to approach the category problem especially considering the idea that 'Recent' is sub-category of all genre categories. Another way to achieve the effect of songs belonging to more than one category is to create an inheritance hierarchy whereby some categories are children of others and this is discussed next.

3.6.3. Mixing categories like colours

In much the same way that every shade of colour is made up from primary colours, mixing two or three categories can create every specific sub-set of song categories. If instead of saying colours are mixed, they are said to inherit from the primary colours then from this a possible methodology was developed. Songs, as discussed previously, had a stipulation that they must belong to one category; however this one category will can be in turn related through inheritance to other categories. Using an ever more complicated structure, any number of genre/era cross-sections could be developed.

In order to propose a schema for song categories, the 'primary colours' of the structure had to first be defined. It is important to note that this structure was proposed for this specific environment only, for if this system were deployed in another radio station then the structure of categories will most likely be different, however the whole point of developing such a schema is to allow a level of customisation. In sub-section 2.6.1 there was a list of categories, which need to be structured into categories of a similar type. It was proposed to split it up as follows:

Genres:

- Rock
- Dance
- HipHop
- ChillOut
- DnB
- SoulnRnB

Eras:

- Recent
- Classics

Descriptive Categories:

- Pop

- Alternative

It is important to note that Genres, Eras and Descriptive categories are not in themselves categories; in fact they are in fact types of category. The reason it was decided to define them as part of the schema was to allow a better control of how categories can be combined, otherwise the number of combinations would not only be very large and confusing, but many would not make sense (such as a category 'ChillOutDnB', the combination of two generally mutually exclusive genres as ChillOut is very slow music and Drum and Bass is extremely high-paced music). The categories that belong to the category types 'genres', 'descriptive categories' and 'eras' can be thought of as the 'primary colours'. Combining two categories from two of the category types can create 'Secondary colours', and 'tertiary colours' are created by mixing two 'secondary colours'. The production rules of this process were summarised like so:

Secondary Colours:

```
GenreEra -> Genre + Era (e.g. RockRecent -> Rock + Recent)
DescatEra -> Descat + Era (e.g. PopRecent -> Pop + Recent)
GenreDescat -> Genre + Descat (e.g. RockPop -> Rock + Pop)
```

Tertiary Colours:

```
GenreDescatEra -> GenreEra + DescatEra
GenreDescatEra -> GenreDescat + DescatEra
GenreDescatEra -> GenreEra + GenreDescat
```

It is important to point that the use of '+' is purely notational only and although its use implies that this is a union operation, it is in fact not, in fact it is the inter-section of the two parents as illustrated by this venn diagram.

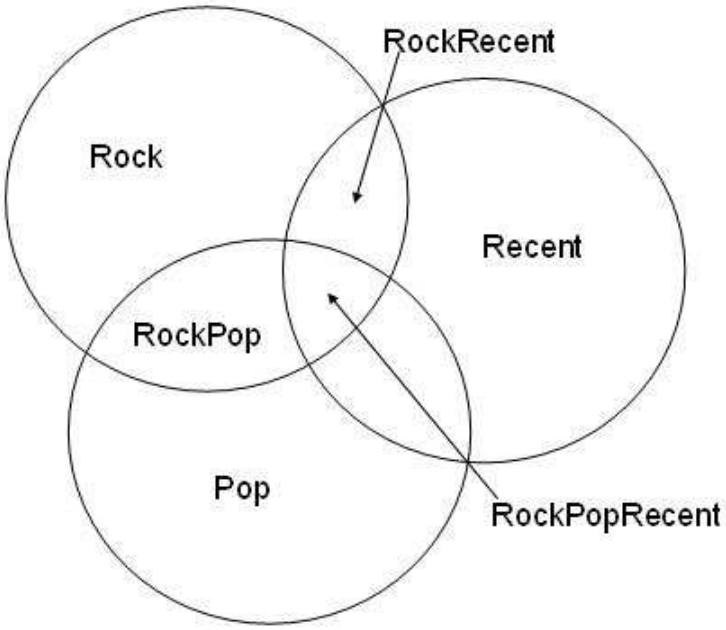
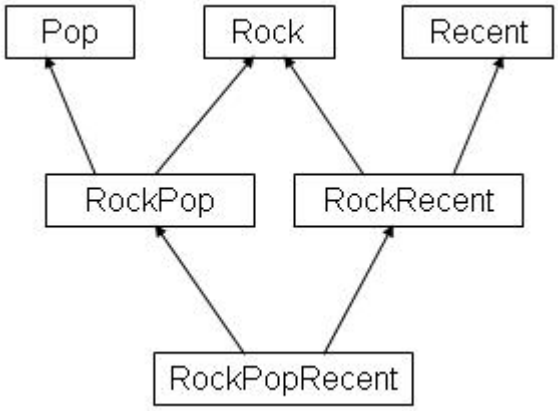


Figure 3.2: Venn diagram showing how primary categories can be mixed

Note in the production rules defined previously how tertiary colours are produced, as it would appear to be much simpler to say that `GenreDescatEra -> Genre + Era + Descat`, but this was not an advisable way to produce it. The reason being that just having this structure would lose the transitivity of the category inheritance in that the tertiary colour 'RockPopRecent' should inherit from not only the primary categories 'Rock', 'Pop' and 'Recent' but also the secondary categories 'RockPop', 'PopRecent' and 'RockPop'. To achieve this, there should be direct inheritance from the secondary categories so that inheritance from the primaries is achieved transitively.



When applying these production rules to the previous defined set of categories elicited from the expert user, it was found that there were 62 different permutations including the original primary colour categories. This was derived manually and the full set can be found in the appendix C.

The purpose behind this abstraction was to provide a flexible structure to categorise songs without the danger of allowing the user too much freedom to let songs belong to multiple categories, which may be logically incompatible.

3.6.4. Requirements elicited

- Each song must belong to one category
- Categories should be able to be defined as sub-categories or one or more other categories
- Songs which belong to a category x must also belong to a category y where y is a parent of x; the songs would also belong to category z where z is the parent of y.
- Categories with no parents shall be defined as primary categories
- Categories that transitively inherit from three primary categories should directly inherit from categories with two primary categories as parents
- Given a category the system must be able to determine the children and parents

3.7. Category limiting

3.7.1. Structured scheduling

The main advantage with having the notion of categories and sub-categories is to allow a much more ordered structure to the scheduling process. Radio stations need to tailor their content to their target audience, and the music they play is integral to this targeting. The targeting differences do not just occur between stations, even within a station the music targeting will need to vary for different times of the day. Using BBC Radio 1 as a case study, the music they play during the daytime is vastly different to the music they play in the evening. The daytime is generally mainstream, recent and popular music that could be found in any high street record shop whereas evening and night-time shows tend to focus on music that is up-and-coming or just obscure. URB follows much the same ethos in that during daytime it is recent and mainstream, and at night the music is either more obscure or it focuses on one genre. Focussing on the daytime music policy of a radio station elicits another important aspect, the station's play-lists. A 'play-list' is normally a restricted list of songs that are new releases and the station normally plays these songs many more times than it plays other categories. This is because these are the songs that the listeners are most likely to want to hear and record companies are very keen to push for more airplay in order to promote their new releases. This means that the system must be able to limit not only what categories that songs can get recommended from, but also how many from each. It is important to ensure that even though play-list tracks may get played more, they still must obey the repetition requirements. In addition to this, it was also vital to certify that there is always an even spread of categories so that there is enough variety within the rigid scheduling structure. It is not a good scheduling technique to recommend songs all from one category and then moving on to another category.

3.7.2. How the existing system addresses this

Analysing the system URB Non-stop, it too has adopted a category-based scheduling method. The system uses hour-file schemas to define what each hour of scheduling should be structured as. These schemas explicitly state one by one the order of categories from which a song will be chosen. It simply is a file structured like so:

```
Recent
Classics
Rock
Recent
Classics
Rock
…
```

This method satisfies the requirements albeit in a slightly implicit fashion. The categories will be evenly spread, there will be a limited number from each category and certain categories will not be played simply. Although computationally this makes the recommender engine easier to develop, the extra burden on the user and the potential for human error is an unacceptable

compromise. It also makes the system vulnerable to serious errors, as David Mayo reported that if someone, for instance, renamed the 'Rock' directory to 'Guitars' without updating the hour file schemas, the system would think that the category 'Rock' had no songs in it and then would be unable to schedule any songs.

3.7.3. Requirements Elicited

- The system must allow the user to specify which categories are allowed to be used to recommend a song from
- The system must allow the user to restrict the number of songs from each category that can be played
- The categories the songs belong to should be as evenly spread as possible

3.8. Coping with errors

3.8.1. Mismatching preferences

As with any scheduler system that relies on a human user setting the preferences, there is the potential for the user to provide rules that limit what is allowed to be played to the extent that the system will fail to recommend a song. In the worse case scenario the rules specified by the user may be logically disjoint leaving no songs available for selection.

3.8.2. Play-out engine dealing with corrupt files

Earlier, the mismatch between metadata and data was said to be a potential cause for the play-out engine attempting to play a song, which no longer exists or at least not in the same place. This problem of attempting to play a file that is unplayable extends to the possibility that the file itself is corrupt. These reasons led to an important requirement that the play-out software should not assume that every file it is scheduled to play is playable and that it should deal with the eventuality of unplayable files.

3.8.3. Requirements elicited

- Must be robust and must continue to operate even if given bad or conflicting rules by the admin user
- The play-out engine should not assume that every song it is scheduled to play will actually be able to be played so should be able to deal with them should they arise.

3.9. Scheduling features

3.9.1. Background

A radio station play-out system is not just about playing songs, often there are other features that need to be played as well, including the news on the hour and

the adverts. The reason features were needed to be considered when designing the scheduler is because entities such as the news are fixed features meaning that the time it is due to play is non-negotiable. The feature must be played at the time stated and so this should dictate what songs should be allowed before the said feature is played. There is another type of feature, a so-called 'soft' feature whereby the timing of the feature is indicative only. Using URB as a case study, the adverts that are played twenty minutes past the hour and twenty minutes to the hour do not have to be played exactly on time, the timings are only indicative only and there is no real need to get them perfect. This is different to the news as listeners would expect the news to be on time, and so in that respect the timings must be exact.

3.9.2. How the existing system addresses this

The system non-stop has the news facility hard-coded into its core programming meaning that if the news slots change (which can happen for example in time of war); the actual non-stop code would have to be modified. A further problem with non-stop is that it does not consider the length of songs before the news at all. Indeed in David Mayo's report there was one incident where the song "Liam Lynch – United States of Whatever" was scheduled to play three seconds before the news, and so played those three seconds and cut to the news. This type of incident can be very frustrating to the listener and would make the radio station sound very unprofessional therefore it is wise to reduce the impact of this where possible. To do this, all the new system needs to do is consider the length of songs it is going to play before the news and make sure that it attempts to schedule a song of appropriate length so minimise or eliminate the cutting of songs.

3.9.3. Requirements elicited

- Must ensure features are played at approximately on time (if a soft feature) or exactly on time (if a fixed feature)

3.10. Summary

These are the raw requirements elicited by analysing the existing system URB non-stop and interviewing the expert user David Mayo. In the next section the requirements will be analysed to validate them and to resolve any conflicts.

4. Requirements Analysis

The core requirements were elicited but it was then important to ensure that there were no contradictions and if there were how they could be mitigated. This section is all about how the clashing requirements can be mitigated by considering their relative priorities and this mitigation gave rise to further requirements.

4.1. Recommending songs when rules clash

As was found in the elicitation process, there was a potential conflict between following the user defined rules and ensuring that a song always get played as it might be the case that the user rules do not match causing there to be no songs being valid. These requirements were potentially in direct discord so it was the case of evaluating the priority of both requirements. Following the rules was an important requirement, but ultimately it was these very rules being in error that was causing this conflict and furthermore the importance of playing a song was of highest priority. It was clear that in case of bad rules causing no songs being recommended, there should be a back up plan so that the system recommends a song even if this song violates one or many of the user defined rules.

In order to know which rules to undo first, it was best to define a rules hierarchy in order of precedence.

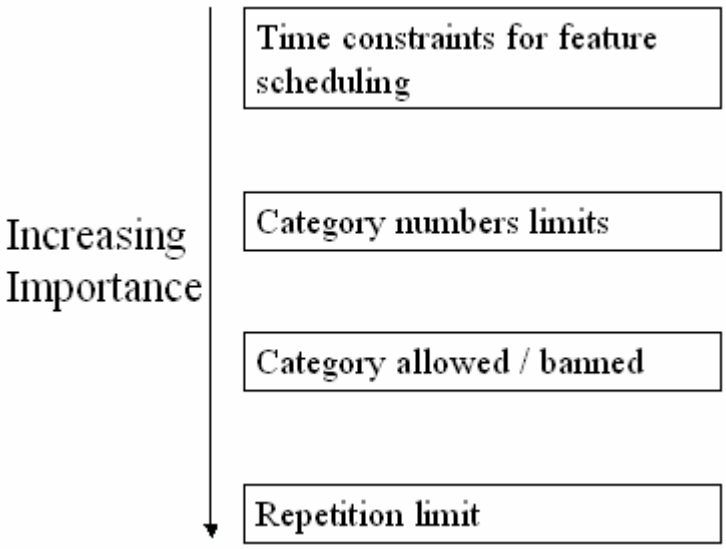


Figure 4.0: Rule precedence hierarchy

As can be seen from Figure 4.0, the lowest precedence is finding matches by time to fill a gap before a fixed feature. This was a relatively unimportant rule as this limit is only applied to make the output sound more professional. After this there were the category rules, firstly the rules that limited the categories by numbers and then the rule that just limited the categories regardless of numbers. Finally in the hierarchy was the rule that prevented songs being repeated, this was the last

rule to get broken but the chances of reaching that far were slim. The repetition limit rule would only get violated if the total song length of the database were less than the repetition limit set by user. It was proposed to edit the requirement so that a song should always be played with the additional statement that said "even if the choice does not quite match with the user's preferences". In addition to this there was a new requirement that rules should be undone in accordance with the rules precedence hierarchy.

4.1.1. Requirements additions/edits

- Must be robust and must continue to operate even if given bad or conflicting rules by the admin user even if the choices go against some of the user's preferences
- Where the rules are such that no song will be recommended, the rules shall be undone in accordance with the rules precedence hierarchy.

4.2. On-the-fly generation and song corruption

It was found in the requirement elicitation that it was necessary to calculate and recommend songs on the fly in order to allow listeners to request songs, unfortunately this requirement is slightly in conflict with the safeguards against unplayable songs. This is because the time between the play-out engine reporting that the song it was due to play is corrupt and the recommender engine recommending another song could be of the order of a several seconds which for a radio station was unacceptable. What was proposed was to introduce new requirements which created a songs-to-play queue system. Instead of the recommender engine just requesting one song ahead, it was decided that the recommender engine should monitor a queue and recommend as many songs as is required to make the songs-to-play queue of sufficient length to ensure that even if one or two of the songs are indeed corrupt, there are enough songs in the queue to absorb the pressure. This queue would be no less than six minutes long in terms of combined song length and no less than three items long so that it is almost certain that if the play-out engine is given a corrupt file, there will always be another file next in the queue that can be played in lieu. Where the queue fails to satisfy both criteria, songs will be recommended until they both hold true. This design drew parallels with the existing system which predetermines an hour's worth of songs in a playlist, the only difference was that this design would only ever predetermine approximately six minutes worth of songs thus allowing requests to be scheduled, albeit not immediately. The fact that requests would not be played immediately was an acceptable trade-off, since there would always be a delay even if a live presenter took the request.

4.2.1. Requirements additions/edits

- Whenever a song is requested, it shall be added to a songs-to-play queue.
- The songs-to-play queue should never be less than six minutes in combined song length and no less then three items long.

- Where the songs-to-play queue fails to meet the minimum song/item length criteria, additional songs will be requested until both criteria are met.

4.3. Applying the user-rules to listener requests

One of the first rules of DJ-ing in clubs is never play a request if it sounds out of place compared to the rest of the music being played, and this rule applies to radio stations as well. Listeners are often unaware of the radio stations intentions and even on a specialist hard trance and dance show, people will still request "Queen – We will rock you" which would sound horrendously out of place. It was important to note that the system would want to satisfy the listener's request but only if it makes sense in the context. Therefore it was only sensible to stipulate an edit to the listener request requirement by adding the caveat that this request would only be accepted if it obeyed the user-defined rules.

4.3.1. Requirements additions/edits

- The system shall be able to take listener requests as long as the number of requests has not exceeded the request limit and also if this request obeys the user-defined rules.

4.4. Ensuring category spread when prioritising the least recently played

It was found that there may be a minor conflict between the requirements that state that priority should be given to those songs that have been least recently played and that songs should be played from a even spread of categories. The reason there may be a conflict was due to the fact that the all the least recently played songs may belong to one category and therefore giving priority to these songs would violate the even spread of categories restriction. The resolution of this conflict laid in the fact that the system should only give *priority* to these songs, not definitely play these songs first. On the matter of category spreading, however, it was much more important to ensure a good variety within the boundaries of the rules as both the casual short-term listener and the long-term listener would get bored if music of the same ilk were to be played consecutively. There was no need to change any requirements but this conflict had to be considered in the design stage.

4.5. Music metadata versus system efficiency

This was only a potential conflict but one worth mentioning since the method of music metadata could have a massive effect on the speed and efficiency of the song recommendation process. This is because all song decisions are based on the music metadata meaning that all files will need to be accessed. It cannot be overemphasised how important it is to ensure that songs are recommended in a time that has no risk of causing a play-out bottleneck where there are no songs in

the songs-to-play queue because the process cannot keep pace with the actual song play-out itself. Therefore if the requirement that all song decisions should utilise the song metadata stored in the datafile compromises the requirement for a real-time system, then the implementation must be mitigated in favour of the real-time requirement. At this stage there was no need to edit or add to the requirements list on this matter, but this had to be considered in later sections of this document.

4.6. Summary

There is now a full set of requirement from which to being to design a system. Section 3 and section 4 have revealed that the common theme it seems is the notion that the system must not fail and if errors are detected then the system must fail safe. This applies also to the efficiency of the system which again is of highest priority. A list of requirements can be found in the Appendix A.

5. Design

5.1. Introduction

The challenges when designing a play-out system like this is how to resolve the sometimes conflicting rules that may arise. This problem can be simplified by making sure the system is designed using an evolutionary life-cycle model as this imitates how rules are naturally layered on top of each other. The requirements analysis set the 'last played' attribute as one of the primary metrics by which a song should be selected from a subset of songs and it is the user defined rules that determine which songs are in the subset or not. This proposed method of selecting from subsets draws many parallels with applying filters to a database table and provides the rationale behind using an evolutionary model, since the more filter criteria this system provides the more structured the song selection will be (see sub-section 2.7.1, structured scheduling).

5.1.1. System overview

From the requirements, it was made clear that a modular approach to the system design was needed. In many ways the ethos of the system was to develop a small-scale multi-system whereby the different components communicate and display autonomous qualities. Indeed, this project was introduced as a part of the future of the fully-automated radio station and agent orientated would take a centre stage in this whole. Moving on from the system as a whole, the first part of the design will focus on how the system was designed from the ground up.

5.2. Primary Design Stages

Using an evolutionary approach, the design of the recommender engine can now conceptually be split into two distinct algorithms:

1. Given an arbitrary set of songs (or the super-set of all songs) select a song for recommendation
2. Given the user defined rules, select a set of songs

Algorithm one represents the fundamental algorithm which was the basis of the recommender engine. It required as an input a set of songs, but for the first stages of design the input was assumed to be, for simplicity's sake, the set of all songs. Algorithm two added structure to how songs are recommended and later on in the design process fed into algorithm one. Having algorithm one on its own would create a 'free for all' situation, but by combining it with algorithm two the design allowed the administrator user greater customisation and limitation of songs thus tailoring the output to suit the radio station's output (see sub-section 2.7.1, structured scheduling).

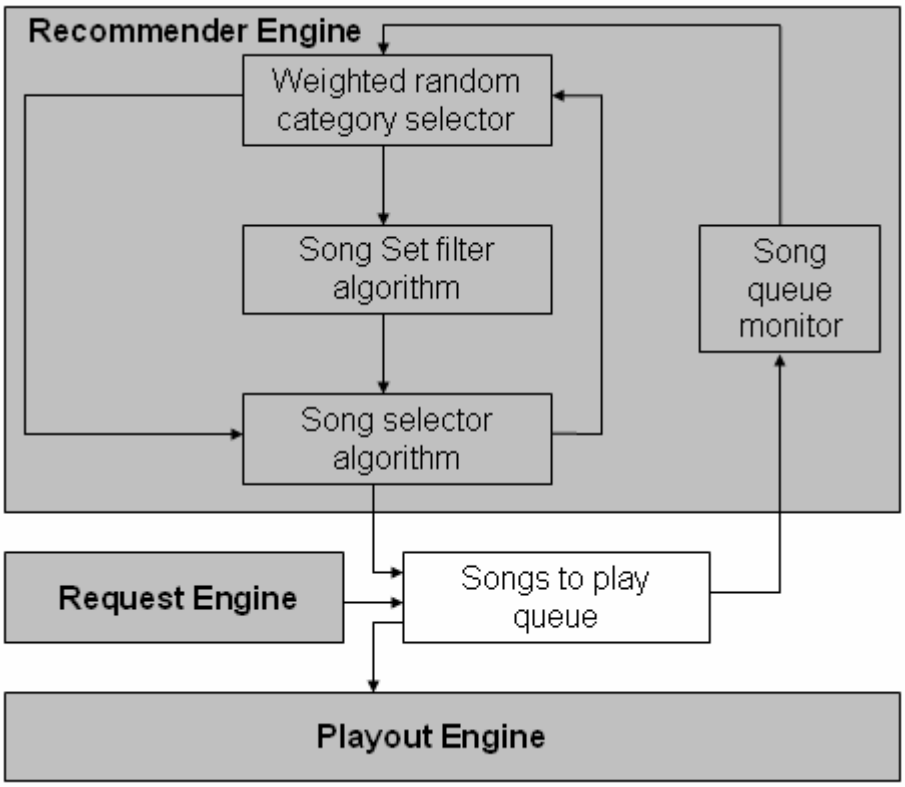


Figure 5.0: Basic flow diagram showing the relationship between the components

The above flow diagram shows the different modules that make up the system, and one by one will be addressed by this design section. Firstly the song selector algorithm will be discussed followed by the song set filter algorithm and how it is layered on top.

5.3. The song selector algorithm

As mentioned before, the purpose of this part of the design is to select a song from a given set therefore this is the base of the recommender engine. The fundamental assumption is that this algorithm is the set of songs given contains only songs allowed by the rules, there will be no checking that songs are of the right category or when the songs were played. The grounding behind this design decision was requirement 7.1 that states:

"The system must be robust and must continue to operate even if given bad rules by the admin user even if the choices are slightly mismatched from user's desires".

This was important, because the system could not assume that the user rules will be consistent, indeed they may even be logically disjoint (see sub-section 3.1, rule clashing) and in that instance the input set for this algorithm would be the empty set. Even at this early stage this failure had to be considered, therefore this algorithm must not check whether the input set of songs obey the rules, because as an emergency measure when encountering an empty input set, the recommender

engine may choose to run this algorithm with the super-set (all songs) so that a song will get chosen.

5.3.1. Deterministic versus random behaviour

Given that all songs in the input set are assumed as valid, there are no 'wrong answers' when deciding which to choose, which is why a simple algorithm that selects a random song from this set would work. Indeed, requirement 3.1 which demands that all song decisions should not be entirely deterministic would appear to support a random approach. It would have been a simple method to implement, but it would not have taken into account requirement 4.4 which stipulates that priority be given to songs played the longest time ago. The solution of the conflict between requirement 3.1 and requirement 4.4 laid in the knowledge that the method of selecting a song from a set forms only part of the entire recommendation process. If the process by which category rules are applied to the filtered set, which is fed to the song selector algorithm, has some random element to it, then requirement 3.1 need not be applied to this algorithm specifically. Since one the primary metric to decide between songs was the attribute 'last played', the best design of this algorithm was to order the set of songs by that attribute and select the song which has been played the longest time ago. This approach then takes into account requirement 3.1 and does not violate requirement 4.4 although a new design requirement had to be established to ensure that the set limiting methods, carried out at the higher level, had to have a randomising element to them.

In summary, this algorithm given a set of songs as an input selected the song least recently played, it then updated its 'last played' attribute and added the song to the list of songs to be played next (the songs-to-play queue).

5.4. Limiting the set of songs for selection by category

This was the first waypoint of the design process, and at this point the recommender engine would be able to decide on songs to play, albeit in a simplistic way because the song selector algorithm was only able to use the universal set of all songs. This was really in essence more of a glorified queue than a song recommender system. The next thing to address was the actual set that is fed into the song selector algorithm, for instead of selecting a song based on all the songs, the song should be selected based on a set that represents the valid songs (i.e. songs that obey the user's rules).

5.4.1. Song queues and sub-queues

Until now, the list from which a song would be recommended has been described as a set but the proposed song selector algorithm conceptually changed this. In effect, the input set of songs to the algorithm was no longer a set of songs; it operated more like a queue of songs. Furthermore due to user rules (such as limiting categories) there would be in fact any number of sub-queues (see figure 5.1) in much the same way as there were any number of sub-sets.

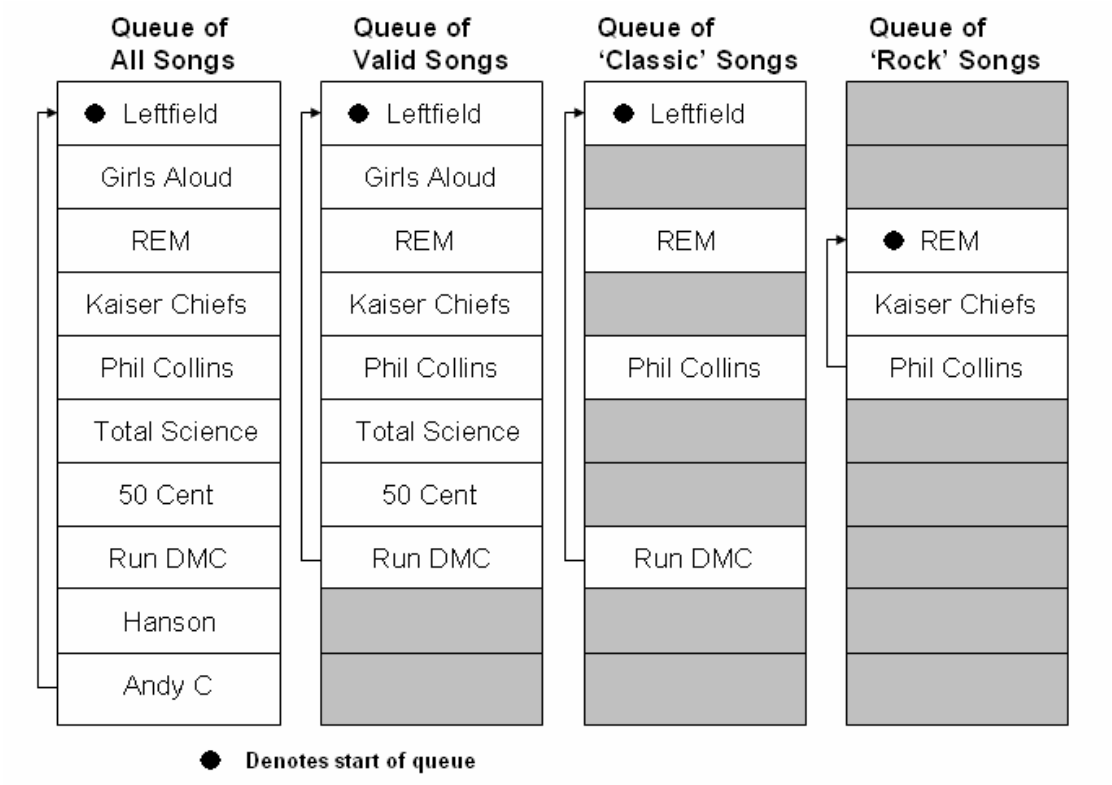


Figure 5.1: Showing how song queues and sub-queues are related

It is worth noting that in this case the 'Queue of valid songs' means the queue of songs (regardless of category) that have not been more recently played than the user-defined repetition threshold (see sub-section 2.5, repetition of songs). Not repeating songs within the time limit was the primary rule of this system and in fact by adding this filter rule to the queue before the song selection algorithm was the second waypoint of the project.

5.4.2. Filtering song sets by categories

As shown in figure 5.1, many different sub-queues can be generated by applying different constraints on the allowed songs. It is important to note that these sub-queues are not 'generated' in the physical sense, they are only generated conceptually. If the universal queue of all songs had {A, B, C}, by applying a constraint a sub-queue e.g. {A, C} may be created. In the second waypoint, the constraint was to filter out all songs played within the repetition time threshold but this did not add structure to the recommending process, which is why the next stage was to incorporate filters by category.

As demonstrated in figure 5.1, sub-queues overlap and this was explained in the requirements analysis where the structure of categories was discussed (see sub-section 2.6, category inheritance). If a category x was stated as being the sub-category of categories y and z, then a song belonging to x would also belong to y and z by transitive inheritance. This presented significant design implications and

came down to the choice of whether it was better to *filter out* or to *limit to* songs belonging to a list of categories. Suppose this was the structure of the categories:

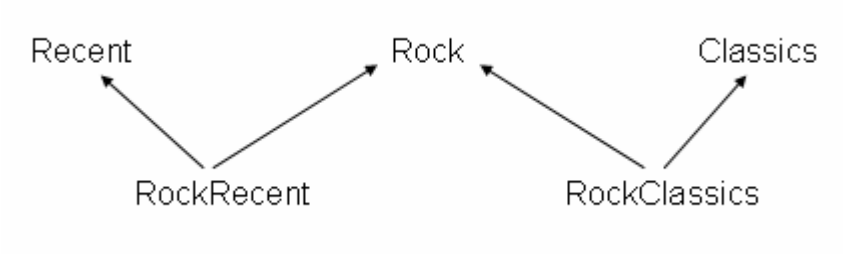


Figure 5.2: An example category structure

As requirement 6.4 states, the system must be able to limit the content to one category. Taking figure 5.2 as an example scenario, how would the system be able to satisfy this requirement given the two methods of applying category restrictions on the song set. The *filter out* proposal would require both categories 'Recent' and 'Classics' to be filtered out, however the problem when this is done is that 'RockRecent' is a child category of 'Recent' so that would also be filtered out, the same applies to 'RockClassics'. This would create a situation whereby songs that had 'Rock' as its category would be included in the set but not songs that were 'RockRecent' or 'RockClassics' which is a conceptual falsehood since these songs have an equal right to be played. In stark contrast, it would be simple to apply the *limit to* proposal to this scenario by limiting it to 'Rock' and through inheritance both 'RockRecent' and 'RockClassics' would be also allowed but not 'Classics' and 'Recent'. It was clear from this that the *limit to* method provided a better solution, if anything because it was a more positive rule. By stipulating certain categories are allowed rather than disallowed, there was a better chance of maximising the number of valid songs in the set and there is no chance that all categories get banned leading to an empty set of songs.

At this point, the design had achieved the next objective and in fact the system was now able to select the least recently played song from a set of songs restricted by a list of valid categories and upper limit for the 'last played' attribute. It was not enough, however, just to limit the categories as there had to be some way of keeping track of which categories were being played and to ensure the right number of songs from each category gets played.

5.4.3. Limiting each category

An idea of how to do this was obtained from URB Non-Stop. The administrator user would explicitly specify the categories to attempt to recommend songs from. A typical file would read:

```
Recent
Rock
Recent
Recent
HipHop
Recent
```

```
Recent
Dance
Recent
Recent
…
```

The recommender engine would recommend a song from a set limited to 'Recent' followed one from 'Rock' (noting that from the previous section 'Rock' = 'Rock' & 'RockRecent'…) and so on. Although this was the best way making sure of an even spread of categories (requirement 6.3), the major drawbacks were the inflexibility of the structure leading to an inability to cope when the recommendation engine fails, for instance when an empty set of songs is produced when applying the user rules. A better way was to specify how many of each category was allowed to be played and assign a quota to each category. This method had the advantage of giving the administrator user full flexibility to allow some categories (by giving them a non-zero quota) and to ban others (by giving them a zero quota). Every time a song is recommended, the quota of the category it belongs is decremented. If a given quota is decremented to zero, songs from that category would no longer be considered.

5.4.4. Limiting categories by explicit or implicit quotas

Given that there was great scope for categories to be very inter-connected, it would cumbersome to force the user to specify a quota for each category. It made more sense instead to allow some category quotas to be derived from their parents. If a song belongs to category x where x is a sub-category of y, the song also must belong to category y, as this is the inheritance structure that was agreed as a requirement. As this inheritance applies to categories, it was fair to suggest that it also applied to category quotas. In figure 5.2, a small part of the category family tree was shown. It is sensible to set quotas for the primary categories ('Recent', 'Rock' and 'Classics'), but for the sub-categories ('RockRecent','RockClassics') the user is less likely to have a specific preference for how many are played from these categories. In this situation, not setting a quota for the sub-categories would be most appropriate. Despite there not being an explicit quota, there still needed to be a way of limiting the number of songs that can be recommended from these categories. It was therefore decided that such categories should implicitly inherit a quota from its parents. For instance, out of all valid songs, a 'RockRecent' song may be selected but 'RockRecent' has no quota, however its parents 'Rock' and 'Recent' both do therefore the selection is valid. Since both are parents, both explicit quotas have an equal right to be decremented by one (to cost the song to that category) but it would be inappropriate to decrement both, therefore it was decided that the one with the highest quota will be decremented. It is important to note that explicit quotas have a higher precedence than implicit quotas which means that if a category has an explicit quota, it would decrement from this and not even consider the quotas of the parents, even if they do have higher quotas. This is called the song costing algorithm (see Appendix B) and is the same algorithm used to determine which quota should be decremented (or 'costed to') whenever a listener requests a song (as per requirement 3.3).

It was now clear the recommender engine could now be customised to only categories with positive quotas (and their children) then when a song is selected by the song selection algorithm, the songs is then 'costed' to a quota (see Appendix B).

5.5. Recommending songs over time

The recommender engine at this point was doing much of what was required as the system had a fair way of selecting songs from song sets based on the last played attribute, and a way of customising the selection set by category in a quantitive manner. The next thing the design needed to address was the fact that recommending songs is not an isolated process, the behaviour over time of the recommender is just as important. Requirement 6.3 instructs the system to recommend songs from an even spread of categories, but the first set customisation algorithm (see Appendix B in no way explicitly enforced this. It enforced the limitation of categories as per requirement 6.2 by using the quota structure but there is no structure by which categories are explicitly selected.

5.5.1. Applying the design to the playlist scenario

The reason why requirement 6.3 was created was with the play-list scenario in mind. Many radio stations have a play-list which is a list of songs (normally new releases) with the intention of playing these songs fairly frequently. This poses a scheduling problem in that the list of songs is small but the frequency of play (the 'rotation' of songs) is high. The problem this caused is because the rotation is very high; every song in the play-list category will have a 'last played' attribute that is fairly recent. Compare this to songs in category 'Rock', and it is likely that songs in this category will have a 'last played' attribute that is less recent. Song selection is based on the union of all sets of categories that have a positive quota, so assuming that 'Rock' has a quota of 5 and 'Play-list' a quota of 20, then each time a song is recommended it will be from a set that is the union of 'Rock' and 'Playlist'. The problem with this lies in the fact highlighted earlier that the least recently played song from 'Playlist' is probably still going to be more recently played than the most recently played song from 'Rock'. Thinking about this in terms of queues, when ordering the union of the two sets, it is expected that 5 rocks songs will be recommended followed by 20 playlist (see Figure 5.3 below).

Figure 5.3: Showing an uneven spread of scheduled categories

This violated the requirement that over time categories should be scheduled with an even spread. The flaw in this design highlighted the need to create an additional algorithm that explicitly decides on a category first, then apply a song set filter based on this one category which then is fed into the song selector algorithm.

5.5.2. Adopting a top-down approach instead of bottom-up

Essentially what the system was doing at this point was using the quotas to find out which categories (and their children) are allowed, and then applying this as set filter to the song set which is then fed in the song selector algorithm. Although the correct quota is decremented when a song is selected and is 'costed' to a category thus ensuring that categories will not get over-played, there is however no control over spread of category scheduling as highlighted by the scenario described in the sub-section 4.5.1. This approach is best described as a bottom-up approach, as first of all the set of all allowed songs is found, then a song is selected and finally it is costed upwards to a category in order to decrement that category's quota. This approach would work very well when dealing with listener requests, as the listener will request a song and this needs to be costed to a category but when the recommender engine is trying to ensure an even spread of categories this approach fails.

It was decided that in fact what was needed was a top-down approach. Top-down means that the recommender engine will select a category first and then applies the set filter based on that category (and by inheritance its children also) only. A

song will be then selected from this set and the quota of the category that formed the filter criteria will then be decremented. The crucial difference with the top-down approach is that the category is explicitly stated when applying the set filter, meaning in order to ensure an even spread of categories some mechanism is needed that selects a category.

5.5.3. Selecting a category for the top-down method

Referring back to sub-section 4.3 where the song selector algorithm was discussed, a design requirement was set stating that set limiting methods should have a random element to them in order to prevent the system from displaying entirely deterministic behaviour. The category selection algorithm was the most appropriate place to introduce this, as non-deterministic behaviour was required to achieve an even spread. It could not be decided entirely randomly however; this would also not make an even distribution over time as category quotas are generally not equal. For instane, if there five 'Rock' and twenty 'Playlist' quotas and the decision were made randomly each time, then statistically speaking 'Rock' is more likely to run out first, but for an even spread this should not be the case. Instead, it was decided that the selection would not just be random, but a *weighted* random based on quota remaining. Taking the previous scenario, the sum of all the quotas is evaluated, then you generate a random number between one and the sum value inclusive. The category selected will depend on this number, if twenty or below 'Playlist' would be selected, if more than twenty then 'Rock' would be selected. That was an outline of the random category algorithm that was decided, and this was extended to cater for an arbitrary number of valid categories. When a category is selected, and the set filter is applied using this category as a criteria (the other rules that have already been described) it produces a subset from which the song least recently played can be selected by the song selector algorithm. It is important to note that the category quota that gets decremented is decided directly by the category selection algorithm and not by the attribute data of the song. It is possible that the song selected could be 'costed' to any number of different category quotas, but it should always be costed to the parent category that filtered the song set. Figure 5.4 (below) illustrates the problem faced.

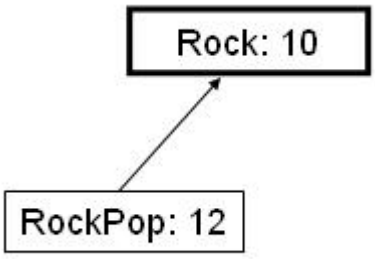


Figure 5.4: Segment of the inheritance tree showing both categories having quotas

In this case, 'Rock' (in bold, on figure 5.4) has been selected as the category, but a song of category 'RockPop' has been selected. This is allowed by the rules

because using inheritance, ‘RockPop’ is a specialised form of ‘Rock’. The situation is confused by the fact that the user has chosen to give explicit quotas to both ‘Rock’ and ‘RockPop’. Which one of the categories should have its quota decremented? If the recommender was evaluating in a bottom-up manner, ‘RockPop’ would have its quota decremented and indeed if this very song were requested by the listener via the request engine then this would have happened, however unlike the request engine, the recommender engine uses the top-down approach. This has the consequence of the ‘Rock’ quota being decremented because it was the set of ‘Rock’that was selected to recommend a song from. Bearing in mind this interpretation, it is irrelevant that the song is also a member of the ‘RockPop’ set, the fact that it is a member of the ‘Rock’ set was the sole reason why this song was considered for selection. On the basis of this reasoning, the recommender engine was designed to decrement quotas based on the category selected and not decrement the quota based on the song selected.

It seemed the proposed category selection algorithm was compatible with what the recommender engine needed, for it was not entirely deterministic in its method but it takes into account quota remaining. The major advantage of this is that it would work effectively over time as every time a song is selected from a set, the category that defined the set has its quota decremented. This means as the category is selected by weighted random numbers; the category just selected would have reduced its chance of getting selected again. Referring back to the situation described earlier (where ‘Play-list’ has a quota of twenty and ‘Rock has a quota of five), at the beginning ‘Play-list’ should have 20/25 chance of getting selected compared to ‘Rock’ which has only a 5/25 chance. Due to the weighted probabilities, ‘Play-list’ should get played more but each time it is, the quota will decrement thus preserving the weighted balance. The spread will not be completely even in that initially ‘Play-list’ will get played more but by the end it should be the case that ‘Play-list’ and ‘Rock’ will have equal chance of being played. It was a minor negative point and it was by far the best method to date of ensuring good variety and balance to the schedule.

5.5.4. A Design running summary

The design at this point had created a well structured recommender engine capable of ensuring the right balance between song-types, as well as giving priority to songs that have been least recently played of the filter song-set. Before the issue of requesting songs and scheduling in advanced features such as the news is addressed, another important element to consider in the design was the robustness of the recommender engine. It was designed with a high-level of flexibility and user-customisation in mind, but with high user-customisation comes also the inevitable possibility of bad instructions. Bad instructions could mean that there are no songs that fit the criteria, so it was clear that some form of backup plan needed to be designed so that the system could cope. This is discussed in detail next.

5.6. Enforcing the rules and guarding against failure

If outside factors (such as hardware failures) are excluded, the actual song selector should never fail unless the input song set it is provided is the empty set, therefore the only modification that was needed for the song selector algorithm was to indicate to the caller function when such a empty set failure has occurred. This took the form of instead of returning the song index, it returned an error code (-1). The potential error really lies in the song set filtering algorithm as whenever the user rules or filter criteria are applied, there is a risk that the set produced is the empty set.

5.6.1. Dealing with an empty song set generated after a category is selected

The empty set will only be produced if the category to filter has no songs belonging to it that have not been played less recently than the user defined repetition limit. This is likely to happen if the category has too few songs for the quota number it has been given. In this situation, a different category must be selected using the weighted random numbers method. To ensure a different category is chosen randonly, the category that produced the failure would be placed temporarily in a banned list so even though it has a non-zero quota the value, the category will not be considered. This was important because it cannot be assumed that only one category with a non-zero quota will produce an empty set error, so if an error occurs again with another category, this too is placed in the banned list and another one is selected. This iterative process ensured that the category chosen will always produce a valid song, and when this happens the temporary banned list is flushed. The rationale behind flushing the temporary banned list is because some categories may be only on this list because its least recently song was 10 seconds away from reaching the repetition allowed threshold and so the next time another song needs to be recommended, the category may now be valid. Just because at some arbitrary time period a category does not produce a valid song is no reason to assume this will always be the case.

5.6.2. Dealing with null category errors

The systematic method of choosing a category, testing whether it produces an empty song set and then banning it and choosing another if it does, solved the no valid songs in a valid category potential failure, but this extra caveat in itself created the potential for another error. Assume there is a situation whereby there are no categories with positive quotas that produce any valid songs; they all produce the empty song set when applying the set filter algorithm. The afore mentioned method will one by one test each category, find there are no valid songs and so appends it to the banned list and tries another. It will get to the situation where there are no more categories to choose and so there is not a valid category. The error would lay in the fact that the weighted random category selector algorithm only produces a null category. This error needed to be caught before the null was applied as a set filter criteria. This is a bad situation to be in, as it would mean that the user rules are now completely incompatible with the current environment and there is no chance that the recommender engine can suggest a song given the boundaries set by the user. This is a terminal situation, so

the recommender engine should resort to a default plan. The most sensible plan would be to apply the song selector algorithm with the an unfiltered song set, and as mentioned earlier in this design section this guaranteed that a song will be selected even if it does not match with the user's desires.

At the same time the recommender engine will also reset all category quotas back to their defaults (the quotas values before they were decremented by song recommendation), in order to guard against the possibility that all categories have been decremented to zero and also to open up the possibility of being able to schedule a song in the standard way the next time a song needs to be recommended. Even if the problem is not rectified by the quota reset, the hierarchy of the error handling will ensure that the system will always resort to the default plan given this error.

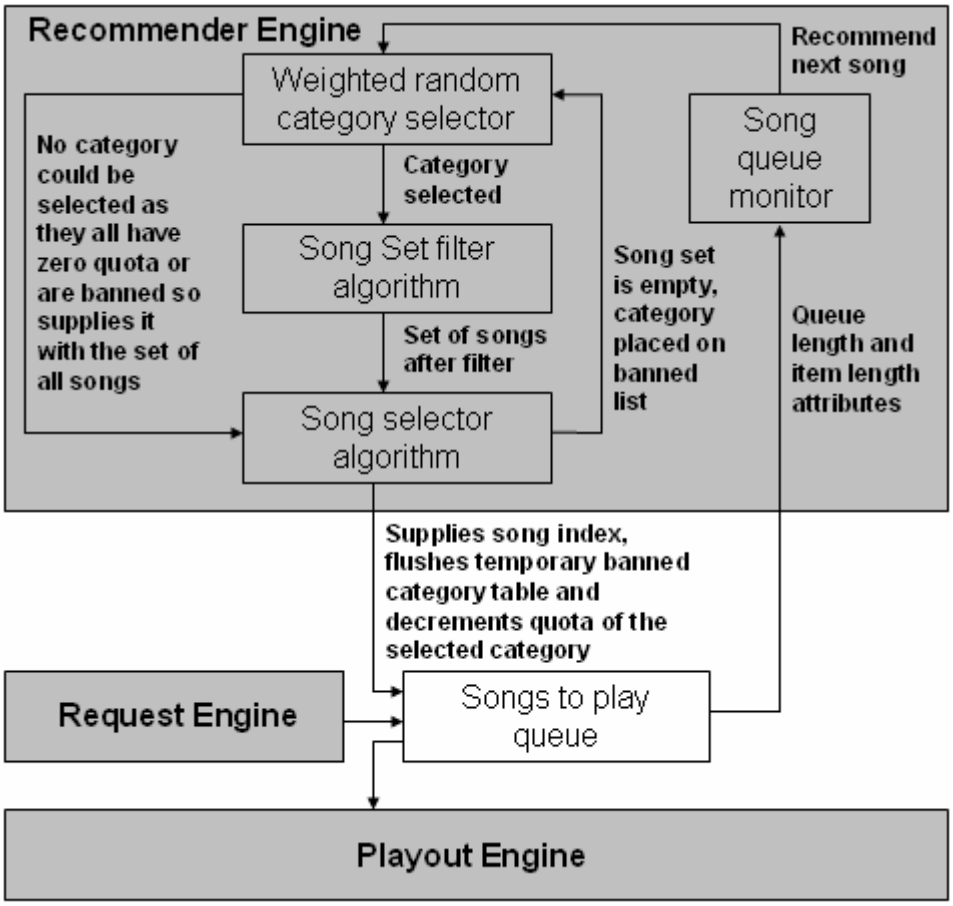


Figure 5.5: An illustration of how errors are handled hierarchically

All errors were designed to be logged, to allow the human user to debug when the error handling method are executed in the hope that the user can rectify the problem.

5.7. Requesting Songs

As seen in Figure 5.5 there is an autonomous section that deals with listener requests. It is worth noting again that all three grey boxes work independently from each other, it is only the song metadata and songs to play queue that is the shared data. The request engine was designed to be simple in operation and takes as an input the song index that the listener wishes to request. This request is not automatically accepted in the same as a computer recommended song, for it must be checked against the rules. First and foremost, the requests have a special quota which is defined by the user, and every time a request is accepted this is decremented. When this reaches zero no more requests will be accepted. Secondly, the song must not have been played within minimum repetition threshold (see requirement 4.1); if it has then the request will not be played. Finally, the song must be successfully costed to a category quota. If the song's category has an explicit quota then it is just the case of decrementing that quota, if the quota is implicit then the bottom-up costing algorithm needs to be applied to attribute the song play to a given category. If the explicit quota is zero or if implicit, all parent categories have a zero-quota, then the request will be rejected.

Whenever a request is accepted, the song index is en-queued (see Appendix B for function) for play-out by the play-out engine. The song is en-queued regardless of the size of the queue to ensure it is played.

5.8. Scheduling features on time

This was the final requirement of the design and represented the most complicated aspect of the scheduler. Features, as was analysed in the requirements section include entities such as adverts and the news. They can be either be 'fixed' (meaning that the feature must be played at the time scheduled exactly) or 'soft' (meaning that the feature must be played only approximately at the scheduled time). Quite obviously, scheduling 'soft' feature is a fairly trivial process but 'fixed' features represent a challenge. In order to schedule a feature exactly on time, the songs preceding must be scheduled with the song length in mind.

5.8.1. Scheduling fixed features

A simple design to deal with this problem would have been just to say that one song before the hard feature is due to play a song of matching length is found to fill the gap. This idea was rejected on the grounds that only thinking one song ahead would run the risk of leaving a gap as little thirty seconds, say, which could not be filled by a song. The recommender engine needs to maximise its chances of being able to schedule a right length song therefore it needs to know about what it has to schedule with. It was decided that scheduling a hard feature on time takes a lower precedence than the repetition rules and allowed category rules therefore the set the algorithm has to select from is no different. The only difference is that instead of just playing the song least recently played from a set filtered by a single category, it will recommend songs based on its length from a set that includes all allowed categories.

This method should begin to operate when the songs-to-play queue end time is within twice the average length of all valid songs. The motivation behind this is that the time left will most likely be able to be filled by two songs or one big song. The algorithm first checks whether the gap can be filled by one song, but if this is not the case then it will systematically check for song pairs that add up the gap remaining. If no exact match is found, it will choose the pair that is slightly too long but is the closest match. In this algorithm, to allow for natural error, an ‘exact match’ was defined as exact or +2 seconds. This tolerance range also gives the algorithm a greater chance of finding a good match.

6. Implementation

6.1. Overview

6.1.1. System structure

The distinct modules in this system, as shown in the design, were segregated to the extent that the only communication between the segments of code is implicit via the shared data sources. In essence they are designed to exhibit behaviour akin to autonomous agents and so it was not unreasonable to suggest that these different modules to the system could in fact be separate programs. The only caveat this would add to the implementation would that the shared data source must exhibit persistent qualities as none of the modules would be sharing any runtime objects or data. From a prototyping perspective, the modular approach made development far easier, as each segment could be considered separately so as long as the interaction with the data store is consistent with the specification, the modules could be overhauled and replaced at will. In order for all the engines to communicate with the shared persistent data, there need to be two platforms, one controlling access to the database which stores the song data and data about quotas and another platform controlling access to the category relationship data. Both platforms have enough methods to view the data and in the case of the category relationship platform, methods to infer assertions to find out who are the parents or children.

6.1.2. Use of existing tools

Due to the scope of the project, there were a number of tools required in the implementation of the system. Most of them are APIs which can be packaged and included as part the system. They will be introduced at this stage, and then later on in the section their inclusion and selection will be justified.

Mp3Info: This was developed by Florian Heer at oeberdosis.de and it provides extensive methods to read and write to ID3V1 and ID3V2 tags. Its source language is written in Java and is a complete library to model and manipulate all types of ID3 tags and frames.

Jena 2: Developed by HP labs it is a library that allows Ontology Web Language schemas and data files to be unified into an inference model. From here, the OWL ontology can be reasoned to get the vital children and parent data of categories.

Mysql-connector-java-3.1.7: Methods to connect to and query mysql databases from java.

Protégé: Very useful ontology development package with the option of converting Protégé format Ontologies into the semantic web standard OWL (Ontology Web Language).

6.1.3. Choice of language

The choice of language was primarily by the current lack of OWL parsing and inferring tools available in any language other than Java, besides the excellent portability of java made it a prime candidate. By choosing Java, incorporating the Jena OWL inference engine was the made much easier, so the choice was a fairly obvious one.

6.2. Metadata storage

6.2.1. Choice of format and how to extract the data

The options of how the music file metadata is stored are limited by requirement 1.2 which requires the metadata to be stored within the same physical file as the music file. As part of the project research, two potential candidates were found to satisfy this, namely ID3 Tags and MPEG-7. MPEG-7 is widely regarded as the future for music metadata (Crysandt & Wellhausen, 2003) but since this technology is very much in its infancy whereas ID3 technology has become virtually a standard for mp3 files it would be prudent to use this rather than MPEG-7. As this system is modular in design, it would be easy to modify the metadata access methods to use MPEG-7 or other music metadata standards at a later date.

The requirements analysis determined what was needed to be stored, and using the ID3 tag specification (Nilsson, 1999) the most suitable tags were mapped to the data storage requirements.

- The artist stored in frame TPE1 (Lead Performers/Soloists, ID3 Version 2)
- The title stored in frame TIT2 (Title/Songname/Content description, ID3 Version 2)
- The album stored in frame TALB (Album, ID3 Version 2)
- The category stored in frame COMM (Comments, ID3 Version 2)
- The year of release stored in frame TYER (Year, ID3 Version 2)
- The length stored in Runtime attribute (ID3 Version 1)

This data is stored within the MP3 file as a tag which is usable by popular music players such as Windows Media Player, Winamp and XMMS to name but a few. The choice of frames was by and large trivial to make apart from the decision to store category in COMM frames instead of the genre frame. The reason behind this was that category is not the same as genre; genre is only one type of category. Also, other applications may make use of the genre frame for their own purposes and considering category may not necessarily mean genre, it would not have been appropriate.

As mentioned in section 5.1.2, one of the many tools that can be used to edit and extract MP3 ID3 metadata is mp3info which is an API written in Java by Florian

Heer. This was chosen due to the ease that the package could be imported into the system and it being implemented in Java meant that it had an intuitive object-orientated structure for ID3 manipulation. In order to use the tool, it was decided that the best way to implement was to create a class called ID3Snapshot, which would act as a platform to extract the relevant ID3 tag frames. This method of implementation fitted in well with the modular ethos of the system, as by adding a platform with a defined output specification, it does not matter whether the underlying metadata format is ID3 or anything else.

```
ID3Snapshot data = new ID3Snapshot(f);
```

'f' is of type java.io.File and is a pointer to the target directory which contains all the mp3s that need processing. The constructor method will parse all the files within the directory and store the music metadata as an array of strings. The way the metadata is extracted is by created ExtendedID3Tag and ID3V2Tag objects and calling methods to extract the required frames. These objects represent a runtime version of the stored data and writing new frames is just as simple as modifying the runtime model and re-writing the tags to the mp3 file.

According to the design, the song selection decisions should be based on the metadata stored within the mp3 file. In theory the idea is good; since metadata stored in a location external to the actual data has a risk that they could become unsynchronised. The problem with the idea is entirely practical in nature as the mp3info software is extremely slow at reading ID3 tags, in fact preliminary testing clocked the process taking an average of eight seconds. Quite clearly this situation would be unacceptable if there were anymore than ten songs, and as the system should be able to cope with upwards of a thousand songs (as per requirement 7.3) then this method could not be implemented. Looking back at the requirements, it was case that real-time efficiency was a more important caveat than ensuring metadata was stored in the same location of data (see section 3.5). Even so, the ID3 tag had to be the primary source of the data and there had to be some mechanism by which data was mirrored in a format that was much more efficient when accessing it. This solved the efficiency concerns but the issue of ensuring a synchronous relationship between the metadata and its mirror was of high importance.

6.2.2. Mirroring for efficiency

As decision-making by the metadata's original source, the ID3 tags, was not an option, there was a free choice in terms of format of the metadata mirror. The decision regarding this had to be taken with the algorithms of the song recommending process in mind (see section 4.3). In the said algorithms the two fundamental processes are 'sort' as in sort the song set descending by the last played attribute, and 'filter' as in filter the song set by category. This mirror must also cope with concurrency as the play-out, recommender and request engines will be sharing this data. It was concluded that a relational database system such as mysql would be ideal for this as it had support for concurrency and is persistent coupled with the ease of sorting and filtering the tables. The process of querying tables to produce data draws many parallels with the theoretical algorithms of

song set filtering and ordering. The requirement that all metadata must be stored in the same location is being preserved, but this data is not being directly addressed when making a recommendation. Even so, the mirror is initially extracted from the metadata, and there are mechanisms in place to ensure that the data is synchronised.

6.2.3. Synchronising the original metadata and the mirror

The problem highlighted before that each mp3 tag read takes approximately 8 seconds means that comprehensive synchronisation can not take place on a very regular basis. Indeed from calculations, a complete synchronisation of one thousand songs could take as much two and a half hours. The system coped much better when it was designed to synchronise the data by parts. That is before a song is confirmed to be recommended, the file is first confirmed to exist and secondly the metadata of the song is extracted and updated to the song list table in the database. In this way the recommender engine ensures that the song is has recommended does exist and that its data is accurate. A similar mechanism was designed to allow the user to add new files to the database, this involved dedicating a special directory where files could be placed and the system would pick up the file, parse it and append it to the database table.

6.3. Database Platform Layer

6.3.1. Choice of language and tools

It made sense that as mysql was the chosen platform for the database of songs the rest of the shared data structures shared the same database. This meant that a single platform class could be defined to control all the shared data. This platform class used a package called mysql-connector which is library that allows a program written in java to connect to and query a database.

In the constructor of the platform class called JBDbase, a connection is created a stored as an object using the following:

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
conn                                                    =
DriverManager.getConnection("jdbc:mysql://localhost/jukebox?user=r
oot&password=XXX");
Statement stmt = conn.createStatement();
stmt.execute("SELECT * FROM SONGLIST");
```

conn is an object variable of an instance of the JBDbase class and will store the details of the connection allowing all platform methods of the instance access the database through the same connection eliminating the connection overhead for each time the database is accessed. In each method of JBDbase that requires mysql access, a statement is created for SQL queries to be run from, and also ResultSet objects and created where it is necessary to extract the data from the query.

```
rs = stmt.getResultSet();
rs.first();
songrec = rs.getInt(1);
```

In this code fragment, the result set is being extracted from the newly executed statement. After this the results of the query can be manipulated at will. In the fragment, it sets the cursor to the first row in the result set and then extracts column one as an integer and stores it into variable songrec. Methods next() and isAfterLast() can be employed in a loop situation if it is the intention to extract all the data from a column from all rows.

6.4. Recommender Engine

6.4.1. Selecting a song from a filtered set

In the design section it was revealed that there were three theoretical algorithms to follow when selecting a song, two of which being the song selector algorithm and the song set filtering algorithm. The song selector algorithm takes a set of songs and simply picks the song with the 'last played' attribute being the least recent, and the song set filtering algorithm just filters out according to category and last played (if it is too recent. Although from a design perspective they are two separate algorithms, from an implementation perspective it is much more effective to combine them. One line of SQL code was enough to satisfy both algorithms.

```
String wherestatement = new String();
if (catchildren!=null) {
       wherestatement = "WHERE (category IN ('" + catchosen + "'";
       for (int i = 0; i < catchildren.length; i++) {
              wherestatement = wherestatement + ", '" + catchildren[i] +
"'";
       }
       wherestatement = wherestatement + ") AND last_played < " +
(System.currentTimeMillis()-3600000) + ")";
}
else wherestatement = "WHERE (category=catchosen)";

...

stmt.execute("SELECT song_id,last_played,rand_seed,category FROM SONGLIST
" + wherestatement + " ORDER BY last_played ASC,rand_seed ASC;");
```

These are two fragments of code from the function *recommendSong()* from the class JBDbase with the top one being the part that generates the where segment of the SQL statement. This fragment uses two variables catchosen and catchildren.String catchosen is evaluated by the output of the weighted random category selector, and catchildren is the array strings containing all the children of catchosen. Catchosen is calculated using the getChildren method of class CatReader which is the class that acts as a platform to access the OWL ontology which describes all the categories and their relationships. As demonstrated, the where statement filters the query and the 'order by' statement sorts the list. It is then just a case of taking the first song_id in the list and then returning that value from the function.

6.4.2. Use of exceptions to guard against failure

In the previous section there was an outline for how songs are selected using mysql based on the original designs in function *recommendsong()* from class JBDbase. There are two modes of failure, one occurs when there are no valid songs for the randomly selected category and another one occurs when there are no valid categories to select.

If a randomly selected category actually had no valid songs to choose, then what would be produced is an empty ResultSet when the statement is executed in mysql. This is where exceptions need to be used to successfully, as by calling the method rs.first() on the ResultSet object rs where rs is an empty set, it would throw a mysql exception. Therefore it was decided to handle this exception and add the category name to the 'banned' table. Additionally, the song_id that is returned is not a valid song_id, it is -1 which is an error code. The reason behind an error code is so that the higher-level class (RecommenderEngine) can then see that an error has occurred in the song recommending process, so it will call the function again.

```
int nextsong == -1;
while (nextsong==-1) {
      nextsong = dbase.recommendNextSong();
}
```

A fragment from RecommenderEngine class, demonstrating the use of iterations to make sure a valid song is selected.

The previous paragraph details what happens when a given valid category is selected but yields no songs, this then adds the category name to the banned list and returns –1 which prompts the higher level caller function to call *recommendSong()* again. This iterative process may produce the second error condition, namely what happens when the random category selector is unable to produce a valid category because either all quotas at zero or all categories are banned.

```
String catchosen = getWeightedRandomCat();
if (catchosen!=null) {
…     // Code which filters the song list and orders it by last
…     // played to attempt to find a song
}
else {
      stmt = conn.createStatement();
      stmt.execute("SELECT  song_id,last_played,rand_seed,category
FROM SONGLIST ORDER BY last_played ASC,rand_seed ASC;");
      rs = stmt.getResultSet();
      rs.first();
      songrec = rs.getInt(1);
}
```

This code fragment shows that a category is found by calling the function *getWeightedRandomCat()* which if there are no valid categories will return NULL. In which case, the system executes its backup plan and recommends a song based on there being no category restrictions. At the same time, all category quotas are refreshed to defaults and the 'banned' table is flushed, this is to maximise the possibility of correct scheduling the next time.

These two methods of error handling mean that the system should never fail even if the user rules mean that there are no valid songs to be played or if there are no categories available to play. This covers both angles of potential major failure, of course in doing this error recovery the choices made by the scheduler may not fit with the user's intended preferences, however this is an acceptable recovery considering it was the user defined rules in the first place that caused the error.

6.5. Category inference

6.5.1. Choice of language

The relationship between the categories was really the crux of this system as the way the categories were configured; affect greatly the interpretation of the songs. Category relationships are effectively a metadata hierarchy, in that metadata describe the song attributes and reveal the category ownership, but there must be a further layer that describes the metadata and add further meaning. What is needed is an ontology that defines each category so that the category relationships can be used as a description logic and triple-based assertions made upon it. As previously researched, OWL (Ontology Web Language) has emerged as the standard for ontologies and is now recommended by the W3 Consortium (see section 2.5). OWL is part of the future of the semantic web and it made sense to use it, the only downside is that support of the technology is still being developed. OWL could only be used to describe the data; it required an inference engine to be able to make assertions on the triples. Jena was found to be suited to this role and had a major advantage in that it is written Java meaning the integration was seamless. Jena also provides an OWL reasoner and many classes to produce runtime models of the ontology, which made it the prime candidate for inclusion.

When creating an ontology in OWL there are two separate files, the OWL Schema and the OWL data. Using much the same terminology as the Object Orientated community, the schema file represents the class definitions and the data file stores the instances. It is the instances that are tested against by the inference engine, however the schema will define the classes the instances belong to this will affect the triples that are outputted. It is the design of the schema that will be addressed first.

6.5.2. Design of the Schema

Ontologies appear at first to be much related conceptually to object orientated programming, however when it comes to realising an ontology in OWL there are a number of important differences. There are two important entities that had to be

considered when designing a schema for an ontology, these are classes and object properties. Object properties are the OWL equivalent of 'instance variables' in java with one crucial difference, they are not defined within the OWL class.

```
<owl:Class rdf:about="cs2ccd:bath-ac-uk:eg/GenreEra" />
<owl:Class rdf:about="cs2ccd:bath-ac-uk:eg/Genre" />
<owl:Class rdf:about="cs2ccd:bath-ac-uk:eg/Era" />

<owl:ObjectProperty rdf:about="cs2ccd:bath-ac-uk:eg/hasGenre">
     <rdfs:range rdf:resource="cs2ccd:bath-ac-uk:eg/Genre" />
     <rdfs:domain   rdf:resource="cs2ccd:bath-ac-uk:eg/GenreEra"
/>
     <rdfs:subPropertyOf           rdf:resource="cs2ccd:bath-ac-
uk:eg/hasParent" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="cs2ccd:bath-ac-uk:eg/hasEra">
     <rdfs:range rdf:resource="cs2ccd:bath-ac-uk:eg/Era" />
     <rdfs:domain   rdf:resource="cs2ccd:bath-ac-uk:eg/GenreEra"
/>
     <rdfs:subPropertyOf           rdf:resource="cs2ccd:bath-ac-
uk:eg/hasParent" />
</owl:ObjectProperty>
```

This fragment defines three classes, "Era", "Genre" and "GenreEra", and also two object properties called "hasGenre" and "hasEra". Note that neither class have any properties associated with them and that the object property is outside all class definitions, they are separate entities. Taking the 'instance variable' analogy, the rdfs:range tag means the type of the variable so 'what is being stored in this property; the rsfs:domain tag means the class this object property belongs; and the rdfs:subPropertyOf tag which is an optional tag that allows you to define an object property as the sub property of another property. The last tag is very important, as it allows properties to be inherited from other property in much the same way as classes can be inherited from other classes; this is explored next.

Referring back to the specific code fragment shown earlier, what this fragment was doing is to define the structure that governs how categories can relate to each other. The property 'hasGenre' and 'hasEra' are both properties of class 'GenreEra' and is the way the physical inheritance link is created between an instance of 'GenreEra' and an instance of 'Genre' and 'Era'. The problem is the requirement of the system was the ability to identify the parents, and these properties are distinct. This is where the rdfs:subPropertyOf tag fits in, as using this extra piece of information it is saying that all 'hasGenre' and 'hasEra' properties are actually all 'hasParent' as well meaning that inferring the property 'hasParent' would succeed in finding all the parents. The classes 'Genre', 'Era' and 'GenreEra' are not categories, they are category types as defined by the requirements elicitation (see section 2.6). 'Genre' and 'Era' are described as primary category types because they have no parents and 'GenreEra' is a secondary category type as it has direct parents but no indirect parents. The final point that needs to be made is to do with transitive inheritance. The 'hasParent' must display transitive properties in order to create links between tertiary

categories (such as ‘RockPopRecent’) and their indirect parents (‘Rock’, ‘Pop’ and ‘Recent’). This effect was achieved by adding the following line of code:

```
<owl:ObjectProperty rdf:about="cs2ccd:bath-ac-uk:eg/hasParent">
                                                    <rdf:type
      rdf:resource="http://www.w3.org/2002/07/owl#Transitive
      Property" />
  </owl:ObjectProperty>
```

This makes the property ‘hasParent’ directly inherit the traits of ‘#TransistiveProperty’ meaning the desired behaviour is achieved.

6.5.3. Creating the data

The data OWL document contains all the instances that the system will require and it was just the case of constructing the document so that all the categories are defined. Firstly start with creating instances of the primary categories

```
<Genre rdf:about="cs2ccd:bath-ac-uk:eg/Rock" />
<Genre rdf:about="cs2ccd:bath-ac-uk:eg/Dance" />
<Genre rdf:about="cs2ccd:bath-ac-uk:eg/HipHop" />
<Genre rdf:about="cs2ccd:bath-ac-uk:eg/ChillOut" />
<Genre rdf:about="cs2ccd:bath-ac-uk:eg/DnB" />
<Genre rdf:about="cs2ccd:bath-ac-uk:eg/SoulnRnB" />
<Category rdf:about="cs2ccd:bath-ac-uk:eg/Pop" />
<Category rdf:about="cs2ccd:bath-ac-uk:eg/Alternative" />
<Era rdf:about="cs2ccd:bath-ac-uk:eg/Recent" />
<Era rdf:about="cs2ccd:bath-ac-uk:eg/Classics" />
```

These are fairly trivial as being primary categories and having no parents themselves, there are no object properties associated with them. Next it was time to create the secondary categories, which are categories that have two parents that are primary categories.

```
<GenreCategory rdf:about="cs2ccd:bath-ac-uk:eg/RockPop">
  <hascGenre rdf:resource="cs2ccd:bath-ac-uk:eg/Rock" />
  <hasgCategory rdf:resource="cs2ccd:bath-ac-uk:eg/Pop" />
</GenreCategory>
```

As is seen here, ‘RockPop’ is defined as having ‘Rock’ as a genre and ‘Pop’ as a category. Both of these object properties are sub-properties of property ‘hasParent’ which means that for property ‘hasParent’ for ‘RockPop’ would return ‘Rock’ and ‘Pop’ which is the desired effect. Finally it is time to define all of the tertiary categories that directly inherit from three secondary categories (which in turn in inherit from three unique primary categories by transitive inheritance).

```
<GenreCategoryEra                    rdf:about="cs2ccd:bath-ac-
  uk:eg/RockPopRecent">
       <hasGenreCategory       rdf:resource="cs2ccd:bath-ac-
    uk:eg/RockPop" />
         <hasGenreEra          rdf:resource="cs2ccd:bath-ac-
    uk:eg/RockRecent" />
      <hasCategoryEra          rdf:resource="cs2ccd:bath-ac-
    uk:eg/PopRecent" />
</GenreCategoryEra>
```

Once again ‘hasGenreEra’ and the other properties all inherit from ‘hasParent’ so that in this case ‘RockPopRecent’ inherits directly from ‘RockPop’, ‘RockRecent’ and ‘PopRecent’, and this in turn means that ‘Rock’, ‘Pop’ and ‘Recent’ are inherited transitively, which was the desired behaviour.

6.5.4. Inferring

The schema and data have now defined the framework and the ontology itself, but this is only useful if the recommender engine is able to query the structure. In order to do this the Jena OWL inference engine was required. Jena is written in java, and the general methodology is to load the schema and data OWL files and Jena converts this ontology into an internal representation of the model.

```
Model schema = ModelLoader.loadModel("OWLSchema.owl");
Model data = ModelLoader.loadModel("OWLData.owl");
Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
reasoner = reasoner.bindSchema(schema);
InfModel infmodel = ModelFactory.createInfModel(reasoner, data);
```

This has now created the model, now the way this model is queried is by using a triples structure. A triple statement will always have a subject, object and predicate. For example, take the statement “RockPop hasParent Rock”. In this instance, ‘RockPop’ is the subject of the statement, ‘hasParent’ is the predicate as it is the property being tested and ‘Rock’is the object. This is a definite statement that would return either true (yes, ‘RockPop’ does ‘hasParent’ ‘Rock’) or false (no, ‘RockPop’ does not ‘hasParent’ ‘Rock’), but when the goal is to find all parents of ‘RockPop’ then testing for true or false in this way is rather a cumbersome way of doing things as all instances would have to be tested against ‘RockPop’. A much cleverer way of doing would be to utilise Jena’s ability to use wildcards when inferring triples. In Jena, subject, predicates or objects can be set to NULL denoting that it is a wildcard so that it will return all statements that match the partial statement supplied. To refer back to the previous example, in order to elicit the parents of ‘RockPop’ would be to infer the following statement: ‘RockPop’ ‘hasParent’ NULL. The object is NULL so that what is returned is not true or false, what is returned is a list of all statements that match this partial statement. The list would consist of:

‘RockPop ‘hasParent’ ‘Rock’
‘RockPop ‘hasParent’ ‘Pop’

Note now that eliciting the parents is just as simple as parsing through the list and extracting all the objects to find all the parents of the category. That is how the parents could be found and finding the children of a category is a similar process. This problem was solved when considering how a child is defined. A child of category x is defined as the child being the subject, the predicate being ‘hasParent’ and the object being category x. The statement that generates the children is very similar to the one that elicits the parents, except the wildcard is in the subject not the object. Again, using ‘RockPop’ has an example but this time

finding the children, the statement would be: NULL ‘hasParent’ ‘RockPop’. Evaluating this generates the following statements:

‘RockPopRecent’ ‘hasParent’ ‘RockPop’
‘RockPopClassics’ ‘hasParent’ ‘RockPop’

So once again parsing the list of statements but this time extracting the subjects will reveal the children of a category.

6.6. Summary

Unfortunately due to the limitations of Jena’s support for OWL at present, the project was unable to make full use of what the OWL language has to offer. Given a bit more support, Jena will be able to support a model of OWL as a mysql implementation. This would be a major change made to the project, as instead of relying on mysql tables, OWL (using mysql as the storage) would be able to store instances of the categories and so store the actual songs in the OWL framework. This would increase the link between the metadata and the ontology. This is not possible now as there is little support for OWL persistent data storage, only runtime storage. Given the separation of the different modules in the system, anything short of persistent data would not work.

7. Evaluation

7.1. Overview

Testing of the system was carried out at every stage of the implementation owing to the way the system evolved from a basic prototype to the comprehensive song recommender system by the end. This section will reveal a selection of the various tests that were carried out to verify that every single module of the system works to their expected specifications.

7.2. Enqueue and dequeue test

Scope: The songs-to-play queue structure

This test is to ensure that mechanism by which songs are placed on the songs-to-play queue works so that the recommender engine can en-queue songs and the play-out simulator can de-queue thus ensuring a safe passage through the shared data structure. The test program is very simple in design and just verifies that songs that are en-queued are de-queued in the correct order, which is the expected output.

```
          JBDbase dbase = new JBDbase();
          System.out.println("Enqueued song 360");
          dbase.enQueue(360);
          System.out.println("Dequeued song " + dbase.deQueue());
          System.out.println("Enqueued song 361");
          dbase.enQueue(361);
          System.out.println("Enqueued song 362");
          dbase.enQueue(362);
          System.out.println("Enqueued song 363");
          dbase.enQueue(363);
          System.out.println("Dequeued song " + dbase.deQueue());
          System.out.println("Dequeued song " + dbase.deQueue());
          System.out.println("Dequeued song " + dbase.deQueue());
          System.out.println("There should be no songs so the next
dequeue command should be in error (-1)");
          System.out.println("Dequeued song " + dbase.deQueue());
```

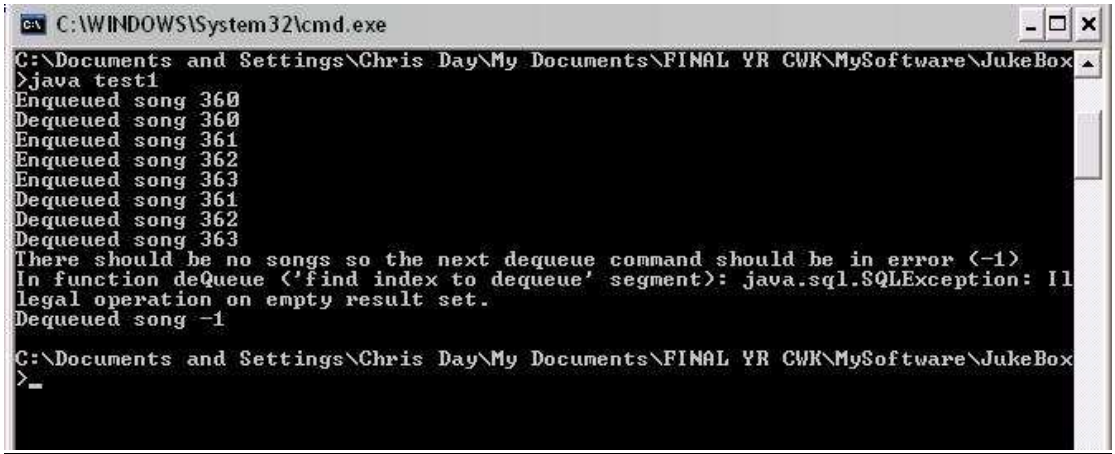


Figure 7.0: Code and screenshot of enQueue and deQueue test results

This is the test program code with result below it. Note that the songs are de-queued in the order that they are en-queued and finally the test program attempts to de-queue when there are no songs to play. This fails, however it fails in a safe way, the program does not execute and returns a song_id –1 that represents the error code. This means that the higher-level program will be able to handle the errors.

7.3. Weighted random category test: 'How random is random'

Scope: Testing the weighted random category selector implementation

There is much debate over the issue of 'random' numbers in computer programs because pure 'random' numbers cannot be generated. Instead, it is a pseudo random which takes a seed (often the system clock) to algorithmically generate a number. The purpose of this test is to evaluate 'how random is random'. This a matter of importance as a biased random could mean that some categories are unfairly favoured more than others, and this is an undesired effect. The algorithm will not produce completely random results as all categories are weighted by the quota they have remaining. For the purposes of testing there are seven categories with the following quotas:

- Recent: 30
- ChillOut: 5
- Dance: 2
- DnB: 1
- Rock: 5
- HipHop: 5
- Pop: 12

The sum of the quotas is sixty and given a large enough sample (this was set at one thousand) the proportion of categories chosen should mirror the proportions of the quotas. This test program will select one thousand random categories which are written to a text file, this is then imported into excel for analysis. It was to produce the following result:

NAME	Quota	Sel from 1000	Quota %	Sel %
Recent	30	508	50.00%	50.80%
ChillOut	5	87	8.33%	8.70%
Dance	2	33	3.33%	3.30%
DnB	1	14	1.67%	1.40%
Rock	5	73	8.33%	7.30%
HipHop	5	85	8.33%	8.50%
Pop	12	200	20.00%	20.00%
TOTAL	60	1000		

The differences between the quota and actual selection percentages are nominal meaning that over time no significant bias is given to any category. This analyses the long-term, but category selection needs to be evenly spread in the short-term too, and the test results did yield some concern over this.

```
HipHop, Pop, Recent, Recent, Recent, Recent, HipHop, Recent,
ChillOut, Recent, Recent, Recent, Recent, Recent, Recent, Recent,
ChillOut
```

This is a fragment of the selected categories and represents probably about an hours worth of music. The point of concern is over the clumping of 'Recent' categories together, which if the system was capable of completely random and even selection should not happen too often. This fragment is one of many where recent is played five times in a row or more. It is most likely due to the fact that random is really only a pseudo-random making it much more prone to localised anomalies as demonstrated here, however further statistical analysis would need to be carried out to confirm this hypothesis.

7.4. Enqueue, dequeue, queue monitoring and concurrency test

Scope: The performance of the songs-to-play queue when accessed by two concurrent programs and the performance of a simple proto-type

The two programs are simple prototypes of the recommender engine and play-out engine and how they are related. The play-out prototype simply de-queues a song every ten seconds whether or not the queue is empty or not. If the queue is empty then –1 will be outputted. The aim is that once both programs are operating, -1 will no longer be outputted and that en-queue and de-queue order is preserved even though the programs are completely separate. The monitor prototype will constantly check the queue every five seconds and if the queue is fewer than three items or less than three-hundred and fifty seconds then songs will be en-queued until the both conditions are untrue. The test was carried out numerous times and not only did the en-queue and de-queue functions perform to specifications, so did the monitor module.

7.5. Finding the parents and the children of a category

Scope: The category parent and child inference engine.

Ensuring that the right parents and children are elicited for a given category is vital to the successful operations of many algorithms in the system therefore it is important to iron out any errors with the category inference. The test program was very simple in design; it just takes an input from the command-line and will output to the screen either all the parents or all the children. This test had to systematically test all sixty-two categories to ensure that there were no errors in the implementation of the OWL schema and data-files. The full test results

revealed a perfect success rate, and although all the results will not be listed here, displayed here is a screenshot of one of them:

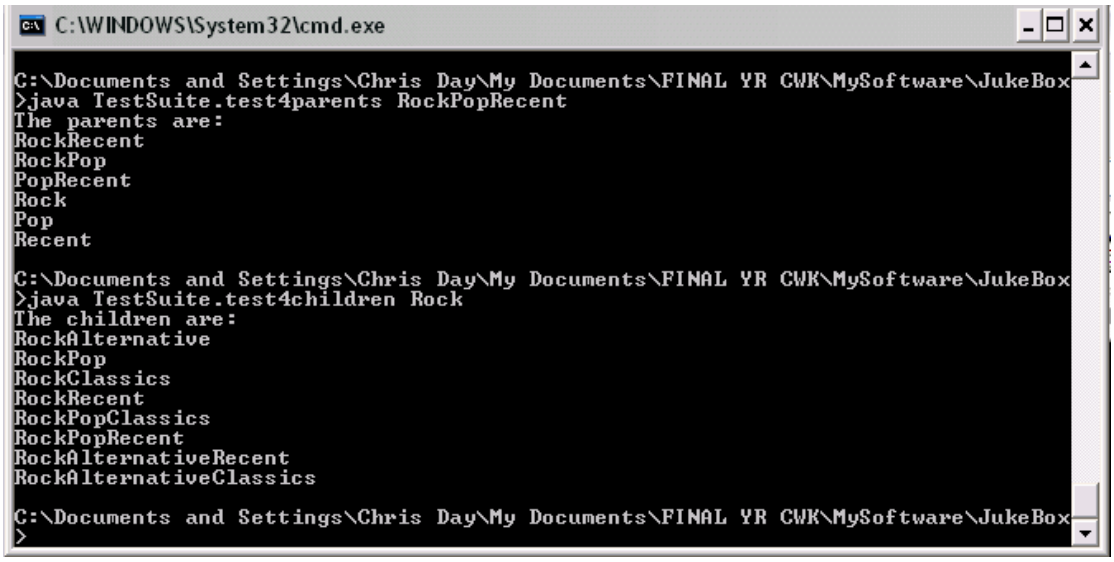


Figure 7.1: Showing the children and parents

Analysing the OWL schema and data file, this is the expected result for both the parents and children and this successful result was reciprocated for all other instances of category.

7.6. Full-system test of stability

Scope: Full test of the robustness of the system. A category with only one song in it is given a quota of five and the play-out engine is playing at twenty-times real time therefore there is a danger of categories running out of songs. The test here is whether the systems fail in a safe state.

It was found in the requirements that the most important aspect of this system is that should a failure occur the system should continue to operate even if the songs chosen are not what the user intended. As the system relies so much on the user's rules making sense, it is imperative that a song will always get recommended when needed. This requires a full system test where all the core components are working together as a whole, as errors will often occur as a result of miscommunication between modules rather than errors within modules. When setting up the test, it was prudent to refer back to the requirements analysis where the potential pitfalls of the system were originally analysed. In this analysis it was revealed that there were two main dangers, firstly that a selected valid category would yield no valid songs to select from and secondly that there were no longer any valid categories to select. Safeguards were put in place, but it was important to establish whether these safeguards were effective or not. In order to test this, a scenario was created whereby the scheduler was inevitably going to face both problems. First off, the 'DnB' category has only one song as part of it, so the quota for 'DnB' was set to five. This would inevitably cause problems, as the

recommender engine will keep suggesting ‘DnB’ as a viable category, yet there are not enough valid songs to satisfy this. Secondly, the play-out engine was modified so that songs were played every twenty seconds regardless of song length. This was primarily because it made testing much faster but also had the convenient side-effect of making time appear to run much faster for the system. The rationale behind this statement is the timestamp for the ‘last_played’ attribute of each song is assigned every time the song is played, however as songs are being played every twenty seconds, the recommender engine is getting through more and more songs. If the average song length is two-hundred seconds, then it appears to the recommender engine’s perception that time is moving ten-times faster than it should. This has the beneficial side-effect of seeing how the system copes when it is rapidly running out of songs to play and therefore running out of valid categories.

The quotas for this test were set at:

- Recent: 20
- ChillOut: 5
- Dance: 4
- DnB: 5
- Rock: 5
- HipHop: 5
- Pop: 12

The test results took the form of the system logs, which is a mysql table that logs every action taken by the system as well as logging all warning it encounters. The expected results were that ‘DnB’ would sooner rather than later play its one song thus reducing its quota down to four but with no other songs to play. Therefore later down the line, the ‘DnB’ category would be selected again but there would be more valid songs to play, so what should happen is another category would be selected and from that new category a song will be selected. As the experiment progresses it was expected that one by one the categories would run out of songs to play until it reaches a point where there were no more categories to select. At this point the system should revert to its default plan and just play the song least recently played, regardless of the category it belongs to.

The experiment was a success, although the entire set of test results will not be placed here, here is a fragment of the system logs that demonstrates the expected behaviour of the system.

6	RECOMMEND	ACTION	Recommended a song from quota: DnB
7	RECOMMEND	ACTION	Recommended song "Total Science – Nosher [Baron VIP Mix].mp3"
8	RECOMMEND	ACTION	Recommended a song from quota: HipHop
9	RECOMMEND	ACTION	Recommended song "The Black Eyed Peas – Shut Up.mp3"
10	PLAY	ACTION	Now Playing: "David Wrench - Superhorny.mp3", length: 210 seconds, category: PopRecent
11	RECOMMEND	ACTION	Recommended a song from quota: Recent
12	RECOMMEND	ACTION	Recommended song "El Presidente - Without You.mp3"
13	PLAY	ACTION	Now Playing: "Total Science - Nosher [Baron VIP Mix].mp3", length: 342 seconds, category: DnB
14	RECOMMEND	WARNING	No valid song found for category: DnB
15	RECOMMEND	ACTION	Recommended a song from quota: Recent
16	RECOMMEND	ACTION	Recommended song "Dogs Die In Hot Cars – Godhopping.mp3"

At line six, the only DnB song is played meaning that subsequent attempts to select a song from DnB fail at line fourteen. At line fifteen, a different category is selected and so the system has recovered from error.

137	RECOMMEND	WARNING	No valid song found for category: Recent
138	RECOMMEND	WARNING	No valid song found for category: DnB
139	RECOMMEND	ACTION	Recommended a song from quota: Pop
140	RECOMMEND	ACTION	Recommended song "Robbie Williams - Millenium.mp3"
141	PLAY	ACTION	Now Playing: "Gouryella - Gouryella.mp3", length: 210 seconds, category: DanceClassics
142	RECOMMEND	WARNING	No valid song found for category: DnB
143	RECOMMEND	WARNING	No valid song found for category: Recent
144	RECOMMEND	WARNING	No valid song found for category: HipHop
145	RECOMMEND	WARNING	All valid categories are producing songs too recently played, recommended song based on <null> category.
146	QUOTA	ACTION	Refreshing quotas back to defaults
147	RECOMMEND	ACTION	Recommended song "Beenie Man - Dude.mp3"
148	PLAY	ACTION	Now Playing: "Crazy Town - Butterfly.mp3", length: 212 seconds,category: RockPopClassics
149	RECOMMEND	WARNING	No valid song found for category: Recent
150	RECOMMEND	WARNING	No valid song found for category: HipHop
151	RECOMMEND	ACTION	Recommended a song from quota: ChillOut
152	RECOMMEND	ACTION	Recommended song "01 Such Great Heights.mp3"

This is approaching the end of the test and valid songs are becoming more and more scarce. It reaches crisis point at line one hundred and forty five when there are no valid categories to select songs from, so song is selected based on <NULL> category meaning that the song was selected from the full list of songs. The quotas are then refreshed to defaults to maximise chances of a recovered, and indeed on line one hundred and fifty-one the system is able to recommend a song based on category again. This was a successful recovery from a potentially fatal situation.

8. Conclusion

8.1. Appraisal

The overall aim of this system was to come one step closer to achieving a well-structured autonomous play-out system for radio stations without the resources to provide live presenter coverage twenty-four hours a day. The innovation in this project lies in the category framework which allows customisation of the station output. It is not just the structure of the output but also the reliability, the ability of the system to cope when the rules it is supplied just do not make sense. At no point should this system fail to recommend a song. The one downside to the project has been the inability to get the news 'hard features' scheduler to work. Theoretically the algorithm that schedules songs before the news should work, but unfortunately due to time constraints, the implementation could not be completed. The system overall design leaves much room for extension as it is important to get this project into context. The play-out software represents an important part of the radio station but there is much potential for greater integration with other system to create the complete radio broadcasting package. These extensions as well as the potential for future work are explored next.

8.2. Extension and future work

This system has provided an extendible framework by which music is categorised by the user and then recommended to play for a radio station play-out engine. The key aspect has been autonomy, as the user sets the rules and the recommender engine will do the rest thus the user is no longer required to explicitly state which category should be selected and when. There is much scope for further work as the greatest goal of all would be to create a completely computerised radio station. The following sections will outline how this project could be extended to come one step closer to achieving this goal.

8.2.1. Voice-tracking

This system can deal with songs and features such as the news but the greatest scheduling achievement would be to allow dynamic "voice-tracking" to take place. Voice tracking is radio jargon term that means a pre-recorded message by the presenter, which is then automatically interspersed with the music by the scheduler. This technique is used by many commercial radio stations that instead of making sure a presenter is live in the studio all the time, there will be periods where the play-out computer will take over and schedule in the pre-recorded voice links at appropriate times. The scheduling challenge is to ensure that the content of the voice message actually makes sense and that the scheduler fulfils the promises made in it. For instance, say the voice message promises that "Moby and Girls Aloud will be playing in the next half-an-hour", then the scheduler will have to store a couple of rules that ensures that indeed these artists are played in the next half-an-hour. Voice-tracking scheduling must take into account the potential

time-specific nature of some pre-recorded messages, since some of them may refer to the time of day, what day of the week or time-dependant facts such as "the news will coming up in ten minutes time". All of these facts mean that often voice-tracking must be implemented in a static and manual fashion, but the ultimate goal would be achieve voice tracking in a completely dynamic fashion whereby the user is taken out of the equation.

To achieve this ultimate goal would require an extension to the current rule hierarchy and to introduce the concept of sequential rules. Sequential rules are not rules that affect any particular instance of a song recommendation, but have a lasting effect on a series of recommendations. To refer back to a previous example, if a voice message is scheduled with the promise that "Moby" and "Girls Aloud" will be played within thirty minutes then these two statements will be added to a list of additional rules or 'promises' as it is more intuitive to think them as. What happens then is the next time a song needs to be recommended then it will first look to the list of 'promises' and see whether any of them can be satisfied and actively look to play either. If the deadline is approaching and still there are rules unsatisfied then this means the other rules are in conflict with the list of promises and so this must be mitigated. The best way to ensure that mitigation is not necessary is to ensure that as a pre-condition to the scheduling of the voice message that the promises can actually be satisfied and to perhaps strategically schedule a different but still valid voice message that does not have promises that are as restrictive. By introducing this facility, the system would be able to claim that it is an autonomous agent-based system capable dynamically running a radio station schedule.

8.2.2. Song classification

Previously with the extension of the scheduler to incorporate actual human voices into the radio station content in a meaningful way, it is close to being a complete and autonomous radio station. This one area that has thus far been neglected is the issue of automatic song classification. This project provides a useful framework in the way categories interact with each other but the ultimate extension would be given an arbitrary set of songs, a method by which songs are automatically classified by their aural characteristics. It is a growing area of research into methods of identifying a song's genre by analysing its waveform. It is very much possible to extend this project's ontology of static song categories and create instead an ontology of characteristics instead. Each characteristic would have a waveform metric and a range that this metric must be within for a song to be said to have this characteristic. Characteristics can then be combined using an ontology to create pseudo-genres of mathematically similar music. These would not be genres in the classical intuitive sense, but they would be mathematically specified genres which given sufficient experimentation and manipulation of the characteristics could make an automatic classification system that is just as good, if not better, than a human musical expert.

8.2.3. Listener request engine

At present the listener request is merely a command line interface, but the fact that it has been written in Java and also the fact that the only interactions this module has with other modules in the system is in the way it manipulates the mysql database means that this code can revamped as a Java Applet or even as a PHP or CGI-perl script allowing the request engine to be hosted on the internet meaning that listeners all around the world can request to hear music on the radio station.

8.2.4. Fulfilling legal requirements

It is a legal requirement for all radio stations to keep copies of all the audio output of the station for 42 days for all holders of a standard UK broadcasting licence obtained from the national licensing body OffCom. There is a system developed that has the ability to carry this task of recording all output (Ffitch & Natt, 2005) but it is unable to communicate with the play-out system at all meaning there is no ability to log when songs have been played and when. Using the song logs created by the play-out system, this data can be shared with system developed by Ffitch & Natt (2005) with the potential to able to extract audio between songs. This has a particular important with application to investigating complaints or extracting presenter vocal links for evaluation purposes because often you do not know an exact time when something is broadcasted but often you will remember when a song is. This is all part of the package that could make up the complete automated radio station, a radio station for the 21st century.

9. Bibliography

[Bechofer et al., 2004] Bechofer S., van Harmelen F., Hendler J., Horrocks I., McGuinness D.L., Patel-Schneider P.F., Stein L.A., (2004) *OWL: Web Ontology Language*, a W3C recommendation found at www.W3.org.

[Berners-Lee et al., 1992] Berners-Lee T., Cailliau R., Groff J-F., Pollermann B., (1992) *World-Wide Web: The information universe*, published in Electronic Networking: Research, Applications and Policy.

[Berners-Lee et al., 2001] Berners-Lee T., Hendler J., Lassila O., (2001) *The Semantic Web*, published in Scientific American, 279.

[Brickley & Guha, 2000] Brickley D., Guha R.V., (2000) *Resource Definition Framework (RDF) Schema Specification 1.0*, a W3C candidate recommendation found at www.W3.org.

[Carroll & Stickler, 2004] Carroll J.J., Stickler P., (2004) *TRiX: RDF Triples in XML*, Technical Report HPL-2003-268, HP Labs.

[Carroll et al., 2003] Carroll J.J., Dickinson I., Dollin C., Reynolds D., Seabourne A., Wilkinson K., (2003) *JENA: implementing the semantic web recommendations*, Technical Report HPL-2003-146, Hewlett Packard Laboratories.

[Chandrasekhan et al., 1999] Chandrasekhan B., Josephson J.R., Benjamins V.R., (1999) *What are Ontologies and why do we need them?*, published in IEEE Intelligent Systems 14(1):pp. 20–26.

[Chen et al., 2003] Chen H., Ding L., Finin T., Zou Y., (2003) *TAGA: Using Semantic Web Technologies in Multi-Agent Systems*, proceedings of the 5th international conference on Electronic commerce.

[Costello et al., 1999] Costello R., Rosenthal A., Seligman L., (1999) *XML, Databases, and Interoperability*, The MITRE Corporation. Federal Database Colloquium, AFCEA, San Diego.

[Crysandt & Wellhausen, 2003] Crysandt H., Wellhausen J., (2003) *Music Classification with MPEG-7*, proceedings: SPIE Storage and Retrieval for Media Databases.

[Decker et al., 2000] Decker S., Melink S., van Harmelen F., Fensel D., Klien M., Broekstra J., Erdmann M., Horrocks I., (2000) *The Semantic Web: the roles of XML and RDF*, published in IEEE Internet Computing, Volume 4, Issue 5.

[Franklin & Graesser, 1996] Franklin S., Graesser A., (1996) *Is it an Agent, or just a Program? A taxonomy for Autonomous Agents*, Proceedings: The Third International Workshop on Agent Theories, Architectures, and Languages.

[Grau, 2004] Grau B.C., (2004) *A possible simplification of the semantic web architecture*, proceedings of the 13th conference on World Wide Web.

[Guessoum & Briot, 1999] Guessoum Z., Briot J-P., (1999) *From active objects to autonomous agents*, published in IEEE Concurrency v. 7, n. 3, p. 68-76.

[Ffitch & Natt, 2005] Ffitch J.P., Natt T.W. (2005) *Recording all Output from a Student Radio Station*, Proceedings: 3rd International Linux Audio Conference.

[Haarslev & Moeller, 2003] Haarslev V., Moeller R., (2003) *RACER: An OWL reasoning agent for the semantic web*, publish location unknown, paper available at http://www.cs.concordia.ca/~haarslev/publications/wi-03.pdf.

[Haustein & Pleumann, 2003] Haustein S., Pleumann J., (2002) *Is Participation in the Semantic Web Too Difficult?*, proceedings: The Semantic Web - ISWC 2002: First International Semantic Web Conference.

[Hendler, 2001] Hendler J., (2001) *Agents and the Semantic Web*, published in IEEE Intelligent Systems, vol. 16, no. 2, Mar./Apr. 2001, pp. 30–37.

[Horrocks et al., 2003] Horrocks I., Patel-Scheider P.F., van Harmelen F., (2003) From *SHIQ and RDF to OWL: The making of a web ontology language*, published in the Journal of Web Semantics, 2003.

[Klyne & Carroll, 2004] Klyne G., Carroll J.J., (2004) *Resource Description Framework (RDF): Concepts and Abstract Syntax*, a W3C Recommendation 10 February 2004, found at www.W3.org.

[Luke et al., 1996] Luke S., Spector L., Rager D. (1996) *Ontology-Based Knowledge Discovery on the World Wide Web*, proceedings: The Workshop on Internet-Based Information Systems, AAAI-96 (Portland, Oregon).

[Nilsson, 1999] Nilsson M. (1999) *ID3 tag version 2.3.0: An informal standard*, published at http://www.id3.org/id3v2.3.0.html

[Nwana, 1996] Nwana H.S., (1996) *Software Agents: An Overview*, published in Knowledge Engineering Review, Vol 11, No 3, pp. 205-244, October/November 1996

[Papadakis & Douligeris, 2002] Papadakis J., Douligeris C. (2002), *Design and architecture of a digital music library on the web*, published in The New Review of Hypermedia and Multimedia, 2002.

[PPL, 2005] Phonographic Performance Limited (2005), *The Webcasting Reciprocal Agreement*, published at http://www.ppluk.com/ppl/ppl_lc.nsf/PDL/LicBroadcasting-Internet?Opendocument

[Spyns et al., 2002] Spyns P., Meersman R., Jarra M., (2002) *Data Modelling versus Ontology Engineering*, SIGMOD Record 31(4),2002,12-17.

[Steels, 1995] Steels L., (1995) *When are robots intelligent autonomous agents?*, published in Robotics and Autonomous Systems, 15:3-9.

[Stone, 1998] Stone P., (1998) *Layered Learning in Multiagent Systems*, published by The MIT Press, ISBN 0-262-19438-4.

[Usdin & Graham, 1998] Usdin T., Graham T., (1998) XML: *Not a Silver Bullet, but great pipe wrench*, ACM Standard View, vol. 6, 1998, pp. 125-132.

[Wooldridge et al., 1999] Wooldridge M., Jennings N., Kinny, D., (1999) *A Methodology for Agent-Oriented Analysis and Design*, presented at Agents '99, Seattle WA.

A. Requirements Appendix

1. Music Metadata

1.1. The music meta data must contain:

1.1.1. The artist

1.1.2. The title

1.1.3. The album

1.1.4. The category

1.1.5. The year of release

1.1.6. The length

1.2. The music metadata used for song decision making must be stored within the music data file to preserve referential integrity

2. System Structure

2.1. Must have a distinct abstraction between the recommender engine, the request engine and the playout simulator

2.2. The song data must be shared available to all modules

2.3. The song data must be persistent

2.4. The song data must allow concurrency

3. Song recommendation and requests

3.1. The song selection process must not be entirely deterministic, there must be a random element to it

3.2. The song recommender engine must operate in realtime and be able to recommend songs faster than it takes to play them

3.3. The system shall be able to take listener requests as long as the number of requests has not exceeded the request limit and also if this request obeys the user-defined rules.

3.4. The system shall recommend songs on a by-need basis only

4. Song repetition

4.1. Songs must not be repeated before the minimum period set by user has elapsed

4.2. Songs by the same artist must not be repeated before the same minimum period has elapsed

4.3. The minimum repetition value must be set based on time rather than number of songs played.

4.4. When recommending songs, priority should be given to songs less recently played

5. Category inference

5.1. Each song must belong to one category

5.2. Categories should be able to be defined as sub-categories or one or more other categories

5.3. Songs which belong to a category x must also belong to a category y where y is a parent of x; the songs would also belong to category z where z is the parent of y.

5.4. Categories with no parents shall be defined as primary categories
5.5. Categories that transitively inherit from three primary categories should directly inherit from categories with two primary categories as parents
5.6. Given a category the system must be able to determine the children and parents

6. Category scheduling and limitation

6.1. The system must allow the user to specify which categories are allowed to be used to recommend a song from
6.2. The system must allow the user to restrict the number of songs from each category that can be played
6.3. The categories the songs belong to should be as evenly spread as possible
6.4. The system must be able to limit to one category

7. Guarding against failure

7.1. Must be robust and must continue to operate even if given bad or conflicting rules by the admin user even if the choices go against some of the user's preferences
7.2. The play-out engine should not assume that every song it is scheduled to play will actually be able to be played so should be able to deal with them should they arise.
7.3. The system should be able to deal with over one thousands songs
7.4. Where the rules are such that no song will be recommended, the rules shall be undone in accordance with the rules precedence hierarchy.

8. Scheduling features

8.1. Must ensure features are played at approximately on time (if a soft feature) or exactly on time (if a fixed feature)

9. Songs-to-play queue

9.1. Whenever a song is requested, it shall be added to a songs-to-play queue.
9.2. The songs-to-play queue should never be less than six minutes in combined song length and no less then three items long.
9.3. Where the songs-to-play queue fails to meet the minimum song/item length criteria, additional songs will be requested until both criteria are met.

B. Code Appendix

This appendix contains the key functions of the project.

B1 recommendNextSong() of class JBDbase.

```
public int recommendNextSong() {
	Statement stmt = null;
	ResultSet rs = null;
	int songrec = -2; //-2 is the error code for 'error while
attempting to recover'
	String catchosen = getWeightedRandomCat();
	if (catchosen!=null) {
		songrec = -1; //-1 is the error code for 'no valid song for
valid cat'
		String[] catchildren = CatReader.getChildren(catchosen);
		String wherestatement = new String();
		if (catchildren!=null) {
			wherestatement = "WHERE (category IN ('" + catchosen
+ "'";
		for (int i = 0; i < catchildren.length; i++) {
		wherestatement = wherestatement + ", '" + catchildren[i] +
"'";
		}
		wherestatement = wherestatement + ") AND last_played < " +
(System.currentTimeMillis()-getRepetitionThreshold()*60*1000) + ")";
		}
		else wherestatement = "WHERE (category=catchosen)";
		try {
		stmt = conn.createStatement();
		stmt.execute("SELECT song_id,last_played,rand_seed,category
FROM SONGLIST " + wherestatement + " ORDER BY last_played ASC,rand_seed
ASC;");
		rs = stmt.getResultSet();
		rs.first(); //If there are no songs, this will throw an
exception which is handled
		songrec = rs.getInt(1);
		stmt.execute("UPDATE cat_quota,categories SET
cat_quota.num=cat_quota.num-1 WHERE cat_name='" + catchosen + "' AND
cat_quota.cat_id=categories.cat_id;");
		stmt.execute("SELECT SUM(num) FROM cat_quota;");
		rs = stmt.getResultSet();
		rs.first();
		if (rs.getInt(1) == 0) {
			appendSysLog("QUOTA","WARNING","All categories now
have quota zero");
			refreshQuotas();
			}
		}
		catch (Exception e) {
			System.out.println("No valid song found for category:
" + catchosen);
			appendSysLog("RECOMMEND", "WARNING","No valid song
found for category: " + catchosen);
			try {
				stmt.execute("INSERT INTO banned (cat_name)
VALUES ('" + catchosen + "');");
			}
			catch (Exception f) {
				System.out.println(f);
			}
```

```
			}
			finally {
				try {
					if (songrec>-1) refreshBannedCats(); //If a
valid song has been selected, delete all banned
					rs.close();
					stmt.close();
				}
				catch (Exception g) {
					System.out.println("In method
recommendNextSong (while closing resources): " + g);
				}
			}
			if (songrec>-1) {
				System.out.println("Just recommended a song from
quota: " + catchosen);
			appendSysLog("RECOMMEND", "ACTION","Recommended a song from
quota: " + catchosen);
			}
		}
		else {
			try {
				stmt = conn.createStatement();
				stmt.execute("SELECT
song_id,last_played,rand_seed,category FROM SONGLIST ORDER BY last_played
ASC,rand_seed ASC;");
				rs = stmt.getResultSet();
				rs.first();
				songrec = rs.getInt(1);
			}
			catch (Exception e) {
				System.out.println("In method recommendNextSong
(recovery from 'no cat' section): " + e);
			}
		System.out.println("All valid categories are producing songs too
recently played, recommended song based on <null> category.");
		appendSysLog("RECOMMEND", "WARNING","All valid categories are
producing songs too recently played, recommended song based on <null>
category.");
		refreshQuotas();
		refreshBannedCats();
		}
		return songrec;
}
```

B2 costToCat() of class JBDbase

```
public int costToCat(String thecat) {
//Given a category name, this function attempts to decrement
//it's quota OR where it's quota is implicit, attempts to
//decrement one of its *parents* quotas.
		Statement stmt = null;
		ResultSet rs = null;
		int returnval = -1;
		try {
			stmt = conn.createStatement();
			String sqlstatement = "SELECT
cat_quota.num,categories.cat_name FROM cat_quota,categories WHERE
categories.cat_name = '" + thecat + "' AND categories.cat_id =
cat_quota.cat_id;";
			stmt.execute(sqlstatement);
			rs = stmt.getResultSet();
			rs.first();
```

```
                int quotaleft = rs.getInt(1);
                if (quotaleft != 0) {
                        returnval = 0; //return value of '0' means
successfully costed
                        sqlstatement = "UPDATE cat_quota,categories SET
cat_quota.num = cat_quota.num - 1 WHERE categories.cat_name = '" + thecat
+ "' AND categories.cat_id = cat_quota.cat_id;";
                        System.out.println(sqlstatement);
                        stmt.execute(sqlstatement);
                }
        }
        catch (Exception e) {
                String[] catparents = CatReader.getParents(thecat);
                String wherestatement = new String();
                wherestatement = "WHERE (categories.cat_name IN ('" +
catparents[0] + "'";
                for (int i = 1; i < catparents.length; i++) {
                        wherestatement = wherestatement + ", '" +
catparents[i] + "'";
                }
                wherestatement = wherestatement + ") AND categories.cat_id =
cat_quota.cat_id)";
                try {
                        stmt = conn.createStatement();
                        String sqlstatement = "SELECT
cat_quota.cat_id,cat_quota.num FROM cat_quota,categories " +
wherestatement + " ORDER BY cat_quota.num DESC;";
                        stmt.execute(sqlstatement);
                        rs = stmt.getResultSet();
                        rs.first();
                        int quotaleft = rs.getInt(2);
                        if (quotaleft != 0) {
                                int catid = rs.getInt(1);
                                //Decrements the quota
                                stmt.execute("UPDATE cat_quota SET num=num-1
WHERE cat_id = " + catid + ";");
                                returnval = 0;
                        }
                }
                catch (Exception f) { //Ignore
                }
        }
        finally {
                try {
                        rs.close();
                        stmt.close();
                }
                catch (Exception g) { //Ignore
                }
        }
        return returnval; //If it returns '0' then it has been costed
successfully
                                        //If it returns '-1' then it was
unable to cost (quota=0)
}
```

B3 enQueue() of class JBDbase

```
public void enQueue(int index) {
/** Inserts a new song in the queue and marks the 'last_played' field of
said song as current system time*/
        int lastindexinqueue = getQueueItemLength();
        Statement stmt = null;
```

```
        ResultSet rs = null;
        try {
                stmt = conn.createStatement();
                stmt.execute("INSERT INTO QUEUE (song_id,num_in_queue)
VALUES ("+index+","+ (lastindexinqueue+1) + ");");
                long currtime = System.currentTimeMillis();
                //Ensures that the same song is not repeated by setting the
last played to current time-stamp
                stmt.execute("UPDATE SONGLIST SET last_played=" + currtime +
" WHERE song_id=" + index + ";");
                //Ensures that the same artist is not repeated as well
                stmt.execute("SELECT artist FROM songlist WHERE song_id=" +
index + ";");
                rs = stmt.getResultSet();
                rs.first();
                String theartist = rs.getString(1);
                stmt.execute("UPDATE songlist SET last_played=" + (currtime-
(getRepetitionThreshold()*30*1000)) + " WHERE artist='" + theartist +
"';");
                if (rs!=null) rs.close();
                if (stmt!=null) stmt.close();
        }
        catch (Exception e) {
                System.out.println("In function enQueue: " + e);
        }
}
```

B4 recommenderEngine (whole class)

```
package uk.ac.bath.cs2ccd.phase3;

import uk.ac.bath.cs2ccd.JukeBoxDatabase.*;

import java.util.Date;

public class RecommenderEngine {

public static void main(String[] args) {

JBDbase dbase = new JBDbase(); //Tag database and queue are updated

int nextsong = -1;

nextsong = doRecommendAndEnqueueSong(dbase);
System.out.println("I've just recommended song '" +
dbase.getFilename(nextsong) + "'");
dbase.appendSysLog("RECOMMEND","ACTION","Recommended song \"" +
dbase.getFilename(nextsong) + "\"");
nextsong = doRecommendAndEnqueueSong(dbase);
System.out.println("I've just recommended song '" +
dbase.getFilename(nextsong) + "'");
dbase.appendSysLog("RECOMMEND","ACTION","Recommended song \"" +
dbase.getFilename(nextsong) + "\"");
nextsong = doRecommendAndEnqueueSong(dbase);
System.out.println("I've just recommended song '" +
dbase.getFilename(nextsong) + "'");
dbase.appendSysLog("RECOMMEND","ACTION","Recommended song \"" +
dbase.getFilename(nextsong) + "\"");
Date daterightnow = new Date(System.currentTimeMillis());
System.out.println("--- [" + daterightnow.toString()+ "]");

while(true) {
```

```
        if (dbase.getQueueItemLength() < 3 || dbase.getQueueTimeLength() <
350) { //Monitors the queue
        nextsong = doRecommendAndEnqueueSong(dbase);
        System.out.println("I've just recommended song '" +
dbase.getFilename(nextsong) + "'");
        dbase.appendSysLog("RECOMMEND","ACTION","Recommended song \"" +
dbase.getFilename(nextsong) + "\"");
        Date rightnow = new Date(System.currentTimeMillis());
        System.out.println("--- ["+rightnow.toString()+"]");
        }
        try {
                Thread.sleep(10000); //Prevents a busy loop
//              Thread.sleep(60000);
        }
        catch (Exception e) {
                System.out.println(e);
        }
}

}

public static int doRecommendAndEnqueueSong(JBDbase dbase) {

int nextsong = -1;

while (nextsong==-1) { //Iterates the song recommend method until valid
one recommended
        nextsong = dbase.recommendNextSong();
}
dbase.enQueue(nextsong);

return nextsong;
}

}
```

B5 catReader (whole class)

```
package uk.ac.bath.cs2ccd.JukeBoxDatabase;

import com.hp.hpl.jena.reasoner.*;
import com.hp.hpl.jena.rdf.model.*;
import com.hp.hpl.jena.util.*;

public class CatReader {

public static String[] getParents(String catname) {

Model schema = ModelLoader.loadModel("OWLSchema.owl"); //Loads the schema
file
Model data = ModelLoader.loadModel("OWLData.owl"); //Loads the data file
Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
reasoner = reasoner.bindSchema(schema);
InfModel infmodel = ModelFactory.createInfModel(reasoner, data);

//By setting p,o,s this is setting up the inference statement
//o is NULL because object is what is being extracted
//p is hasParent
//s is the category in question, so all combine to ask the question
//"What are the parents of catname?"

Property p = infmodel.getProperty("cs2ccd:bath-ac-uk:eg/hasParent");
Resource o = null;
```

```
Model m = infmodel;
Resource s = infmodel.getResource("cs2ccd:bath-ac-uk:eg/" + catname);
int numparents = 0;
String[] parents = new String[255];

for (StmtIterator i = m.listStatements(s,p,o); i.hasNext(); ) {
//Parses all statements and extracts the parents
       Statement stmt = i.nextStatement();
       String obj = stmt.getObject().toString().substring(21);
       parents[numparents] = obj;
       numparents++;
}
String[] retval = null;
if (numparents != 0) {
       retval = new String[numparents];
       for (int i = 0; i < numparents; i++) {
              retval[i]=parents[i];
       }
}
return retval; //returns the parents in a string array
}

public static String[] getChildren(String catname) {

Model schema = ModelLoader.loadModel("OWLSchema.owl");
Model data = ModelLoader.loadModel("OWLData.owl");
Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
reasoner = reasoner.bindSchema(schema);
InfModel infmodel = ModelFactory.createInfModel(reasoner, data);

//Similar to getparents except it is the subject that is NULL
//and o is set to catname
//The question that is being asked is "Who has catname as their parent"

Property p = infmodel.getProperty("cs2ccd:bath-ac-uk:eg/hasParent");
Resource s = null;
Model m = infmodel;
Resource o = infmodel.getResource("cs2ccd:bath-ac-uk:eg/" + catname);
int numchildren = 0;
String[] children = new String[255];

for (StmtIterator i = m.listStatements(s,p,o); i.hasNext(); ) {
       Statement stmt = i.nextStatement();
       String subj = stmt.getSubject().getLocalName();
       children[numchildren] = subj;
       numchildren++;
}
String[] retval = null;
if (numchildren != 0) {
       retval = new String[numchildren];
       for (int i = 0; i < numchildren; i++) {
              retval[i]=children[i];
       }
}
return retval;
}

}
```

C. Category permutations Appendix

Primary Colours	Secondary Colours	Tertiary Colours
Genres		
Rock	RockPop	RockPopRecent
Dance	RockAlternative	RockPopClassics
HipHop	RockRecent	RockAlternativeRecent
ChillOut	RockClassics	RockAlternativeClassics
DnB	DancePop	DancePopRecent
SoulnRnB	DanceAlternative	DancePopClassics
	DanceRecent	DanceAlternativeRecent
Categories	DanceClassics	DanceAlternativeClassics
Pop	HipHopPop	HipHopPopRecent
Alternative	HipHopAlternative	HipHopPopClassics
	HipHopRecent	HipHopAlternativeRecent
Era	HipHopClassics	HipHopAlternativeClassics
Recent	ChillOutPop	ChillOutPopRecent
Classics	ChillOutAlternative	ChillOutPopClassics
	ChillOutRecent	ChillOutAlternativeRecent
	ChillOutClassics	ChillOutAlternativeClassics
	DnBPop	DnBPopRecent
	DnBAlternative	DnBPopClassics
	DnBRecent	DnBAlternativeRecent
	DnBClassics	DnBAlternativeClassics
	SoulnRnBPop	SoulnRnBPopRecent
	SoulnRnBAlternative	SoulnRnBPopClassics
	SoulnRnBRecent	SoulnRnBAlternativeRecent
	SoulnRnBClassics	SoulnRnBAlternativeClassics
	PopRecent	
	PopClassics	
	AlternativeRecent	
	AlternativeClassics	

D. Test Dump Appendix

1	PLAY	WARNING	Checked queue and found 0 items. Player is paused and channel is silent. Next check 20 seconds.
2	PLAY	WARNING	Checked queue and found 0 items. Player is paused and channel is silent. Next check 20 seconds.
3	PLAY	WARNING	Checked queue and found 0 items. Player is paused and channel is silent. Next check 20 seconds.
4	RECOMMEND	ACTION	Recommended a song from quota: Recent
5	RECOMMEND	ACTION	Recommended song "David Wrench - Superhorny.mp3"
6	RECOMMEND	ACTION	Recommended a song from quota: DnB
7	RECOMMEND	ACTION	Recommended song "Total Science - Nosher [Baron VIP Mix].mp3"
8	RECOMMEND	ACTION	Recommended a song from quota: HipHop
9	RECOMMEND	ACTION	Recommended song "The Black Eyed Peas - Shut Up.mp3"
10	PLAY	ACTION	Now Playing: "David Wrench - Superhorny.mp3", length: 210 seconds, category: PopRecent
11	RECOMMEND	ACTION	Recommended a song from quota: Recent
12	RECOMMEND	ACTION	Recommended song "El Presidente - Without You.mp3"
13	PLAY	ACTION	Now Playing: "Total Science - Nosher [Baron VIP Mix].mp3", length: 342 seconds, category: DnB
14	RECOMMEND	WARNING	No valid song found for category: DnB
15	RECOMMEND	ACTION	Recommended a song from quota: Recent
16	RECOMMEND	ACTION	Recommended song "Dogs Die In Hot Cars - Godhopping.mp3"
17	PLAY	ACTION	Now Playing: "The Black Eyed Peas - Shut Up.mp3", length: 222 seconds, category: HipHopPop
18	RECOMMEND	ACTION	Recommended a song from quota: Pop
19	RECOMMEND	ACTION	Recommended song "The Coral - Bill McCai.mp3"
20	PLAY	ACTION	Now Playing: "El Presidente - Without You.mp3", length: 203 seconds, category: RockRecent
21	RECOMMEND	ACTION	Recommended a song from quota: Rock
22	RECOMMEND	ACTION	Recommended song "Rammstein - Asche Zu Asche.mp3"
23	PLAY	ACTION	Now Playing: "Dogs Die In Hot Cars - Godhopping.mp3", length: 156 seconds , category: RockRecent
24	RECOMMEND	ACTION	Recommended a song from quota: Recent
25	RECOMMEND	ACTION	Recommended song "Moby - Lift Me Up.mp3"
26	PLAY	ACTION	Now Playing: "The Coral - Bill McCai.mp3", length: 156 seconds, category : RockPop
27	RECOMMEND	ACTION	Recommended a song from quota: Rock
28	RECOMMEND	ACTION	Recommended song "Hundred Reasons - Silver.mp3"
29	PLAY	ACTION	Now Playing: "Rammstein - Asche Zu Asche.mp3", length: 233 seconds, category: RockAlternative
30	RECOMMEND	ACTION	Recommended a song from quota: ChillOut
31	RECOMMEND	ACTION	Recommended song "Phil Collins - In the air tonight.mp3"
32	PLAY	ACTION	Now Playing: "Moby - Lift Me Up.mp3", length: 193 seconds, category : DancePopRecent
33	RECOMMEND	ACTION	Recommended a song from quota: Pop
34	RECOMMEND	ACTION	Recommended song "Eiffel 65 - Blue (Da ba dee).mp3"
35	PLAY	ACTION	Now Playing: "Hundred Reasons - Silver.mp3", length: 197 seconds, category: Rock
36	RECOMMEND	ACTION	Recommended a song from quota: ChillOut
37	RECOMMEND	ACTION	Recommended song "Black Box Recorder - The Facts Of Life.mp3"
38	PLAY	ACTION	Now Playing: "Phil Collins - In the air tonight.mp3", length: 329 seconds, category: ChillOutClassics
39	RECOMMEND	ACTION	Recommended a song from quota: ChillOut
40	RECOMMEND	ACTION	Recommended song "Plumb - Damaged [Broke Down Palace Soundtrack].mp3"
41	PLAY	ACTION	Now Playing: "Eiffel 65 - Blue (Da ba dee).mp3", length: 218 seconds, category: DancePop
42	RECOMMEND	ACTION	Recommended a song from quota: HipHop
43	RECOMMEND	ACTION	Recommended song "Kid Rock - Cowboy.mp3"

44	PLAY	ACTION	Now Playing: "Black Box Recorder - The Facts Of Life.mp3", length: 273 seconds, category: ChillOutClassics
45	RECOMMEND	ACTION	Recommended a song from quota: Recent
46	RECOMMEND	ACTION	Recommended song "03 - chemicalbrothersremixed.com - Believe (Belief , Elektric Cowboy).mp3"
47	PLAY	ACTION	Now Playing: "Plumb - Damaged [Broke Down Palace Soundtrack].mp3", length: 230 seconds, category: ChillOut
48	RECOMMEND	ACTION	Recommended a song from quota: Dance
49	RECOMMEND	ACTION	Recommended song "Binary Finary - 1999.mp3"
50	PLAY	ACTION	Now Playing: "Kid Rock - Cowboy.mp3", length: 256 seconds, category : HipHopClassics
51	RECOMMEND	WARNING	No valid song found for category: DnB
52	RECOMMEND	ACTION	Recommended a song from quota: Recent
53	RECOMMEND	ACTION	Recommended song "Portobella - Covered In Punk.mp3"
54	PLAY	ACTION	Now Playing: "03 - chemicalbrothersremixed.com - Believe (Belief, Elektric Cowboy).mp3", length: 335 seconds, category: DanceAlternativeRecent
55	RECOMMEND	ACTION	Recommended a song from quota: ChillOut
56	RECOMMEND	ACTION	Recommended song "Lemon Jelly - Come.mp3"
57	PLAY	ACTION	Now Playing: "Binary Finary - 1999.mp3", length: 184 seconds, category: DanceClassics
58	RECOMMEND	WARNING	No valid song found for category: DnB
59	RECOMMEND	ACTION	Recommended a song from quota: Pop
60	RECOMMEND	ACTION	Recommended song "Pink - Missundaztood - 07- Just Like A Pill.mp3"
61	PLAY	ACTION	Now Playing: "Portobella - Covered In Punk.mp3", length: 207 seconds, category: RockAlternativeRecent
62	RECOMMEND	ACTION	Recommended a song from quota: Recent
63	RECOMMEND	ACTION	Recommended song "06-the_crystal_method-realizer-ph.mp3"
64	PLAY	ACTION	Now Playing: "Lemon Jelly - Come.mp3", length: 236 seconds, category: ChillOut
65	RECOMMEND	ACTION	Recommended a song from quota: Recent
66	RECOMMEND	ACTION	Recommended song "Fatboy Slim - The Journey.mp3"
67	PLAY	ACTION	Now Playing: "Pink - Missundaztood - 07- Just Like A Pill.mp3", length: 235 seconds, category: Pop
68	RECOMMEND	ACTION	Recommended a song from quota: Recent
69	RECOMMEND	ACTION	Recommended song "Eminem - Like Toy Soldiers.mp3"
70	PLAY	ACTION	Now Playing: "06-the_crystal_method-realizer-ph.mp3", length: 227 seconds, category: DanceAlternativeRecent
71	RECOMMEND	ACTION	Recommended a song from quota: HipHop
72	RECOMMEND	ACTION	Recommended song "Kelis - Milkshake.mp3"
73	PLAY	ACTION	Now Playing: "Fatboy Slim - The Journey.mp3", length: 275 seconds, category: DanceRecent
74	RECOMMEND	ACTION	Recommended a song from quota: Dance
75	RECOMMEND	ACTION	Recommended song "Simple Kid - Drugs.mp3"
76	PLAY	ACTION	Now Playing: "Eminem - Like Toy Soldiers.mp3", length: 295 seconds, category: HipHopRecent
77	RECOMMEND	ACTION	Recommended a song from quota: Pop
78	RECOMMEND	ACTION	Recommended song "Garbage - I Think im Paranoid.mp3"
79	PLAY	ACTION	Now Playing: "Kelis - Milkshake.mp3", length: 179 seconds, category: HipHopPop
80	RECOMMEND	ACTION	Recommended a song from quota: Recent
81	RECOMMEND	ACTION	Recommended song "Kaiser Chiefs - Oh My God.mp3"
82	PLAY	ACTION	Now Playing: "Simple Kid - Drugs.mp3", length: 212 seconds, category: DanceAlternative
83	RECOMMEND	ACTION	Recommended a song from quota: Dance
84	RECOMMEND	ACTION	Recommended song "York - The Awakening.mp3"
85	PLAY	ACTION	Now Playing: "Garbage - I Think im Paranoid.mp3", length: 215 seconds , category: RockPopClassics
86	RECOMMEND	ACTION	Recommended a song from quota: Pop
87	RECOMMEND	ACTION	Recommended song "Wheatus - Teenage Dirtbag.mp3"
88	PLAY	ACTION	Now Playing: "Kaiser Chiefs - Oh My God.mp3", length: 214 seconds, category: RockRecent

89	RECOMMEND	ACTION	Recommended a song from quota: Recent
90	RECOMMEND	ACTION	Recommended song "Paul Van Dyk - Crush.mp3"
91	PLAY	ACTION	Now Playing: "York - The Awakening.mp3", length: 195 seconds, category: Dance
92	RECOMMEND	ACTION	Recommended a song from quota: Pop
93	RECOMMEND	ACTION	Recommended song "Made In London - Dirty Water.mp3"
94	PLAY	ACTION	Now Playing: "Wheatus - Teenage Dirtbag.mp3", length: 230 seconds, category: Pop
95	RECOMMEND	ACTION	Recommended a song from quota: Rock
96	RECOMMEND	ACTION	Recommended song "U2 - The Sweetest Thing.mp3"
97	PLAY	ACTION	Now Playing: "Paul Van Dyk - Crush.mp3", length: 227 seconds, category : DanceRecent
98	RECOMMEND	WARNING	No valid song found for category: DnB
99	RECOMMEND	ACTION	Recommended a song from quota: Pop
100	RECOMMEND	ACTION	Recommended song "04_daft_punk-harder_better_faster_stronger-wAx.mp3"
101	PLAY	ACTION	Now Playing: "Made In London - Dirty Water.mp3", length: 274 seconds, category: Pop
102	RECOMMEND	WARNING	No valid song found for category: DnB
103	RECOMMEND	ACTION	Recommended a song from quota: Rock
104	RECOMMEND	ACTION	Recommended song "Kid Symphony - Hands On The Money.mp3"
105	PLAY	ACTION	Now Playing: "U2 - The Sweetest Thing.mp3", length: 176 seconds, category : RockClassics
106	RECOMMEND	ACTION	Recommended a song from quota: HipHop
107	RECOMMEND	ACTION	Recommended song "Bubba Sparxxx - Deliverance.mp3"
108	PLAY	ACTION	Now Playing: "04_daft_punk-harder_better_faster_stronger-wAx.mp3", length: 223 seconds, category: DancePopClassics
109	RECOMMEND	ACTION	Recommended a song from quota: Recent
110	RECOMMEND	ACTION	Recommended song "osymyso-Its All About Fun, Right.mp3"
111	PLAY	ACTION	Now Playing: "Kid Symphony - Hands On The Money.mp3", length: 149 seconds, category: RockAlternative
112	RECOMMEND	ACTION	Recommended a song from quota: Pop
113	RECOMMEND	ACTION	Recommended song "Bonnie Tyler - I need a Hero (Footloose Soundtrack).mp3"
114	PLAY	ACTION	Now Playing: "Bubba Sparxxx - Deliverance.mp3", length: 247 seconds, category: HipHop
115	RECOMMEND	ACTION	Recommended a song from quota: Pop
116	RECOMMEND	ACTION	Recommended song "Levellers - Just The One.mp3"
117	PLAY	ACTION	Now Playing: "osymyso-Its All About Fun, Right.mp3", length: 132 seconds, category: DanceAlternativeRecent
118	RECOMMEND	ACTION	Recommended a song from quota: ChillOut
119	RECOMMEND	ACTION	Recommended song "Sugababes - Shape.mp3"
120	PLAY	ACTION	Now Playing: "Bonnie Tyler - I need a Hero (Footloose Soundtrack).mp3" , length: 348 seconds, category: PopClassics
121	RECOMMEND	ACTION	Recommended a song from quota: Pop
122	RECOMMEND	ACTION	Recommended song "Chuck Berry - Johnny B. Goode.mp3"
123	PLAY	ACTION	Now Playing: "Levellers - Just The One.mp3", length: 165 seconds, category: PopClassics
124	RECOMMEND	WARNING	No valid song found for category: DnB
125	RECOMMEND	ACTION	Recommended a song from quota: Rock
126	RECOMMEND	ACTION	Recommended song "Adam Green - Jessica.mp3"
127	PLAY	ACTION	Now Playing: "Sugababes - Shape.mp3", length: 249 seconds, category : ChillOutPop
128	RECOMMEND	WARNING	No valid song found for category: Recent
129	RECOMMEND	ACTION	Recommended a song from quota: Dance
130	RECOMMEND	ACTION	Recommended song "Gouryella - Gouryella.mp3"
131	PLAY	ACTION	Now Playing: "Chuck Berry - Johnny B. Goode.mp3", length: 157 seconds, category: PopClassics
132	RECOMMEND	WARNING	No valid song found for category: HipHop
133	RECOMMEND	WARNING	No valid song found for category: Recent
134	RECOMMEND	ACTION	Recommended a song from quota: Pop

135	RECOMMEND	ACTION	Recommended song "Crazy Town - Butterfly.mp3"
136	PLAY	ACTION	Now Playing: "Adam Green - Jessica.mp3", length: 151 seconds, category: Rock
137	RECOMMEND	WARNING	No valid song found for category: Recent
138	RECOMMEND	WARNING	No valid song found for category: DnB
139	RECOMMEND	ACTION	Recommended a song from quota: Pop
140	RECOMMEND	ACTION	Recommended song "Robbie Williams - Millenium.mp3"
141	PLAY	ACTION	Now Playing: "Gouryella - Gouryella.mp3", length: 210 seconds, category: DanceClassics
142	RECOMMEND	WARNING	No valid song found for category: DnB
143	RECOMMEND	WARNING	No valid song found for category: Recent
144	RECOMMEND	WARNING	No valid song found for category: HipHop
145	RECOMMEND	WARNI3NG	All valid categories are producing songs too recently played, recommended song based on <null> category.
146	QUOTA	ACTION	Refreshing quotas back to defaults
147	RECOMMEND	ACTION	Recommended song "Beenie Man - Dude.mp3"
148	PLAY	ACTION	Now Playing: "Crazy Town - Butterfly.mp3", length: 212 seconds, category: RockPopClassics
149	RECOMMEND	WARNING	No valid song found for category: Recent
150	RECOMMEND	WARNING	No valid song found for category: HipHop
151	RECOMMEND	ACTION	Recommended a song from quota: ChillOut
152	RECOMMEND	ACTION	Recommended song "01 Such Great Heights.mp3"
153	PLAY	ACTION	Now Playing: "Robbie Williams - Millenium.mp3", length: 232 seconds, category: Pop
154	RECOMMEND	WARNING	No valid song found for category: Recent
155	RECOMMEND	ACTION	Recommended a song from quota: Rock
156	RECOMMEND	ACTION	Recommended song "Ian Drury - Sex and Drugs and Rock and Roll.MP3"